Oracle PL/SQL语言初级教程

PL/SQL 语言基础

PL/SQL 是一种高性能的基于事务处理的语言，能运行在任何 ORACLE 环境中，支持所有数据处理命令。通过使用 PL/SQL 程序单元处理 SQL 的数据定义和数据控制元素。

- Oracle PL/SQL语言基础(1)
- Oracle PL/SQL语言基础(2)
- Oracle PL/SQL语言基础(3)

复合数据类型

PL/SQL 有两种复合数据结构：记录和集合。记录由不同的域组成，集合由不同的元素组成。在本文中我们将讨论记录和集合的类型、怎样定义和使用记录和集合。

- 复合数据类型(1)
- 复合数据类型(2)
- 复合数据类型(3)
- 复合数据类型(4)
- 复合数据类型(5)

单行函数和组函数

函数是一种有零个或多个参数并且有一个返回值的程序。在 SQL 中 Oracle 内建了一系列函数，这些函数都可被称为 SQL 或 PL/SQL 语句，函数主要分为两大类：单行函数和组函数。

- 单行函数和组函数详解(1)
- 单行函数和组函数详解(2)
- 单行函数和组函数详解(3)
- 单行函数和组函数详解(4)
- 单行函数和组函数详解(5)

表和视图

Oracle 中表是数据存储的基本结构。Oracle 中引入了分区表和对象表，视图是一个或多个表中数据的逻辑表达式。本文我们将讨论怎样创建和管理简单的表和视图。

- 表和视图(1)
- 表和视图(2)

完整性约束

完整性约束是一种规则，不占用任何数据库空间。完整性约束存在数据字典中，在执行 SQL 或 PL/SQL 期间使用。用户可以指明约束是启用的还是禁用的，当约束启用时，他增强了数据的完整性，否则，则反之，但约束始终存在于数据字典中。

- 完整性约束(1)
- 完整性约束(2)
- 完整性约束(3)

过程和函数

过程和函数都以编译后的形式存放在数据库中，函数可以没有参数也可以有多个参数并有一个返回值。过程有零个或多个参数，没有返回值。函数和过程都可以通过参数列表接收或返回零个或多个值，函数和过程的主要区别不在于返回值，而在于他们的调用方式。

- 过程和函数(1)
- 过程和函数(2)

操作和控制语言

SQL 语言共分为四大类：数据查询语言 DQL，数据操纵语言 DML， 数据定义语言 DDL，数据控制语言 DCL。其中用于定义数据的结构，比如 创建、修改或者删除数据库；DCL 用于定义数据库用户的权限。

- 数据操作和控制语言详解(1)
- 数据操作和控制语言详解(2)
- 数据操作和控制语言详解(3)

游标

当执行一条 DML 语句后，DML 语句的结果保存在四个游标属性中，这些属性用于控制程序流程或者了解程序的状态。当运行 DML 语句时，PL/SQL 打开一个内建游标并处理结果，游标是维护查询结果的内存中的一个区域。

- 游标使用大全(1)
- 游标使用大全(2)
- 游标使用大全(3)

异常处理

PL/SQL 处理异常不同于其他程序语言的错误管理方法，PL/SQL 的异常处理机制与 ADA 很相似，有一个处理错误的全包含方法。

- 异常处理初步(1)
- 异常处理初步(2)

PL/SQL 语言基础

Oracle PL/SQL 语言基础(1)

PL/SQL 是 ORACLE 对标准数据库语言的扩展，ORACLE 公司已经将 PL/SQL 整合到 ORACLE 服务器和其他工具中了，近几年中更多的开发人员和 DBA 开始使用 PL/SQL，本文将讲述 PL/SQL 基础语法，结构和组件、以及如何设计并执行一个 PL/SQL 程序。

PL/SQL 的优点

从版本 6 开始 PL/SQL 就被可靠的整合到 ORACLE 中了，一旦掌握 PL/SQL 的优点以及其独有的数据管理的便利性，那么你很难想象 ORACLE 缺了 PL/SQL 的情形。PL/SQL 不是一个独立的产品，他是一个整合到 ORACLE 服务器和 ORACLE 工具中的技术，可以把 PL/SQL 看作 ORACLE 服务器内的一个引擎，sql 语句执行者处理单个的 sql 语句，PL/SQL 引擎处理 PL/SQL 程序块。当 PL/SQL 程序块在 PL/SQL 引擎处理时，ORACLE 服务器中的 SQL 语句执行器处理 pl/sql 程序块中的 SQL 语句。

PL/SQL 的优点如下：

. PL/SQL 是一种高性能的基于事务处理的语言，能运行在任何 ORACLE 环境中，支持所有数据处理命令。通过使用 PL/SQL 程序单元处理 SQL 的数据定义和数据控制元素。

. PL/SQL 支持所有 SQL 数据类型和所有 SQL 函数，同时支持所有 ORACLE 对象类型

. PL/SQL 块可以被命名和存储在 ORACLE 服务器中，同时也能被其他的 PL/SQL 程序或 SQL 命令调用，任何客户/服务器工具都能访问 PL/SQL 程序，具有很好的可重用性。

. 可以使用 ORACLE 数据工具管理存储在服务器中的 PL/SQL 程序的安全性。可以授权或撤销数据库其他用户访问 PL/SQL 程序的能力。

. PL/SQL 代码可以使用任何 ASCII 文本编辑器编写，所以对任何 ORACLE 能够运行的操作系统都是非常便利的

. 对于 SQL，ORACLE 必须在同一时间处理每一条 SQL 语句，在网络环境下这就意味作每一个独立的调用都必须被 oracle 服务器处理，这就占用大量的服务器时间，同时导致

网络拥挤。而 PL/SQL 是以整个语句块发给服务器，这就降低了网络拥挤。

PL/SQL 块结构

PL/SQL 是一种块结构的语言，组成 PL/SQL 程序的单元是逻辑块，一个 PL/SQL 程序包含了一个或多个逻辑块，每个块都可以划分为三个部分。与其他语言相同，变量在使用之前必须声明，PL/SQL 提供了独立的专门用于处理异常的部分，下面描述了 PL/SQL 块的不同部分：

声明部分(Declaration section)

声明部分包含了变量和常量的数据类型和初始值。这个部分是由关键字 DECLARE 开始，如果不需要声明变量或常量，那么可以忽略这一部分；需要说明的是游标的声明也在这一部分。

执行部分(Executable section)

执行部分是 PL/SQL 块中的指令部分，由关键字 BEGIN 开始，所有的可执行语句都放在这一部分，其他的 PL/SQL 块也可以放在这一部分。

异常处理部分(Exception section)

这一部分是可选的，在这一部分中处理异常或错误，对异常处理的详细讨论我们在后面进行。

PL/SQL 块语法

[DECLARE]
---declaration statements
BEGIN
---executable statements
[EXCEPTION]
---exception statements
END

PL/SQL 块中的每一条语句都必须以分号结束，SQL 语句可以使多行的，但分号表示该语句的结束。一行中可以有多条 SQL 语句，他们之间以分号分隔。每一个 PL/SQL 块由 BEGIN 或 DECLARE 开始，以 END 结束。注释由--标示。

PL/SQL 块的命名和匿名

PL/SQL 程序块可以是一个命名的程序块也可以是一个匿名程序块。匿名程序块可以用在服务器端也可以用在客户端。

命名程序块可以出现在其他 PL/SQL 程序块的声明部分，这方面比较明显的是子程序，子程序可以在执行部分引用，也可以在异常处理部分引用。

PL/SQL 程序块可背独立编译并存储在数据库中，任何与数据库相连接的应用程序都可以访问这些存储的 PL/SQL 程序块。ORACLE 提供了四种类型的可存储的程序：

. 函数

. 过程

. 包

. 触发器

函数

函数是命名了的、存储在数据库中的 PL/SQL 程序块。函数接受零个或多个输入参数，有一个返回值，返回值的数据类型在创建函数时定义。定义函数的语法如下：

```
FUNCTION name [{parameter[,parameter,...])] RETURN datatypes IS
[local declarations]
BEGIN
execute statements
[EXCEPTION
exception handlers]
END [name]
```

过程

存储过程是一个 PL/SQL 程序块，接受零个或多个参数作为输入(INPUT)或输出(OUTPUT)、或既作输入又作输出(INOUT)，与函数不同，存储过程没有返回值，存储过程不能由 SQL 语句直接使用，只能通过 EXECUT 命令或 PL/SQL 程序块内部调用，定义存储过程的语法如下：

```
PROCEDURE name [(parameter[,parameter,...])] IS
[local declarations]
BEGIN
execute statements
```

[EXCEPTION
exception handlers]
END [name]

包(package)

包其实就是被组合在一起的相关对象的集合，当包中任何函数或存储过程被调用，包就被加载入内存中，包中的任何函数或存储过程的子程序访问速度将大大加快。

包由两个部分组成：规范和包主体(body),规范描述变量、常量、游标、和子程序，包体完全定义子程序和游标。

触发器(trigger)

触发器与一个表或数据库事件联系在一起的，当一个触发器事件发生时，定义在表上的触发器被触发。

PL/SQL 语言基础

Oracle PL/SQL 语言基础(2)

变量和常量

变量存放在内存中以获得值，能被 PL/SQL 块引用。你可以把变量想象成一个可储藏东西的容器，容器内的东西是可以改变的。

声明变量

变量一般都在 PL/SQL 块的声明部分声明，PL/SQL 是一种强壮的类型语言，这就是说在引用变量前必须首先声明，要在执行或异常处理部分使用变量，那么变量必须首先在声明部分进行声明。

声明变量的语法如下：

Variable_name [CONSTANT] databyte [NOT NULL][:=|DEFAULT expression]

注意:可以在声明变量的同时给变量强制性的加上 NOT NULL 约束条件，此时变量在初始化时必须赋值。

给变量赋值

给变量赋值有两种方式：

. 直接给变量赋值

```
X:=200;
Y=Y+(X*20);
```

. 通过 SQL SELECT INTO 或 FETCH INTO 给变量赋值

```
SELECT SUM(SALARY),SUM(SALARY*0.1)
INTO TOTAL_SALARY,TATAL_COMMISSION
FROM EMPLOYEE
WHERE DEPT=10;
```

常量

常量与变量相似，但常量的值在程序内部不能改变，常量的值在定义时赋予，，他的声明方式与变量相似，但必须包括关键字 CONSTANT。常量和变量都可被定义为 SQL 和用户定义的数据类型。

```
ZERO_VALUE CONSTANT NUMBER:=0;
```

这个语句定了一个名叫 ZERO_VALUE、数据类型是 NUMBER、值为 0 的常量。

标量(scalar)数据类型

标量(scalar)数据类型没有内部组件，他们大致可分为以下四类：

. number

. character

. date/time

. boolean

表 1 显示了数字数据类型；表 2 显示了字符数据类型；表 3 显示了日期和布尔数据类

型。

表 1 Scalar Types:Numeric

Datatype	Range	Subtypes	description
BINARY_INTEGER	-214748-2147483647	NATURAL NATURAL NPOSITIVE POSITIVEN SIGNTYPE	用于存储单字节整数。 要求存储长度低于 NUMBER 值。 用于限制范围的子类型(SUBTYPE): NATURAL:用于非负数 POSITIVE:只用于正数 NATURALN:只用于非负数和非 NULL 值 POSITIVEN:只用于正数，不能用于 NULL 值 SIGNTYPE:只有值:-1、0 或 1.
NUMBER	1.0E-130-9.99E125	DEC DECIMAL DOUBLE PRECISION FLOAT INTEGERIC INT NUMERIC REAL SMALLINT	存储数字值，包括整数和浮点数。可以选择精度和刻度方式，语法： number[（[,]）]。 缺省的精度是 38,scale 是 0.
PLS_INTEGER	-2147483647-2147483647		与 BINARY_INTEGER 基本相同，但采用机器运算时，PLS_INTEGER 提供更好的性能 。

表 2 字符数据类型

datatype	rang	subtype	description
CHAR	最大长度 32767 字节	CHARACTER	存储定长字符串，如果长度没有确定，缺省是 1
LONG	最大长度 2147483647 字节		存储可变长度字符串
RAW	最大长度 32767 字节		用于存储二进制数据和字节字符串，当在两个数据库之间进行传递时，RAW 数据不在字符集之间进行转换。
LONGRAW	最大长度 2147483647		与 LONG 数据类型相似，同样他也不能在字符集之间进行转换。

ROWID	18 个字节		与数据库 ROWID 伪列类型相同，能够存储一个行标示符，可以将行标示符看作数据库中每一行的唯一键值。
VARCHAR2	最大长度 32767 字节	STRINGVARCHAR	与 VARCHAR 数据类型相似，存储可变长度的字符串。声明方法与 VARCHAR 相同

表 3 DATE 和 BOOLEAN

datatype	range	description
BOOLEAN	TRUE/FALSE	存储逻辑值 TRUE 或 FALSE,无参数
DATE	01/01/4712 BC	存储固定长的日期和时间值，日期值中包含时间

LOB 数据类型

LOB(大对象,Large object) 数据类型用于存储类似图像，声音这样的大型数据对象，LOB 数据对象可以是二进制数据也可以是字符数据，其最大长度不超过 4G。LOB 数据类型支持任意访问方式，LONG 只支持顺序访问方式。LOB 存储在一个单独的位置上，同时一个"LOB 定位符"(LOB locator)存储在原始的表中，该定位符是一个指向实际数据的指针。在 PL/SQL 中操作 LOB 数据对象使用 ORACLE 提供的包 DBMS_LOB.LOB 数据类型可分为以下四类：

. BFILE

. BLOB

. CLOB

. NCLOB

操作符

与其他程序设计语言相同，PL/SQL 有一系列操作符。操作符分为下面几类：

. 算术操作符

. 关系操作符

. 比较操作符

. 逻辑操作符

算术操作符如表 4 所示

operator	operation
+	加
-	减
/	除
*	乘
**	乘方

关系操作符主要用于条件判断语句或用于 where 子串中，关系操作符检查条件和结果是否为 true 或 false,表 5 是 PL/SQL 中的关系操作符

operator	operation
<	小于操作符
<=	小于或等于操作符
>	大于操作符
>=	大于或等于操作符
=	等于操作符
!=	不等于操作符
<>	不等于操作符
:=	赋值操作符

表 6 显示的是比较操作符

operator	operation
IS NULL	如果操作数为 NULL 返回 TRUE
LIKE	比较字符串值
BETWEEN	验证值是否在范围之内
IN	验证操作数在设定的一系列值中

表 7.8 显示的是逻辑操作符

operator	operation
AND	两个条件都必须满足
OR	只要满足两个条件中的一个
NOT	取反

执行部分

执行部分包含了所有的语句和表达式,执行部分以关键字 BEGIN 开始，以关键字 EXCEPTION 结束，如果 EXCEPTION 不存在，那么将以关键字 END 结束。分号分隔每一条语句，使用赋值操作符:=或 SELECT INTO 或 FETCH INTO 给每个变量赋值，执行部分的错误将在异常处理部分解决，在执行部分中可以使用另一个 PL/SQL 程序块，这种程序块被称为嵌套块

所有的 SQL 数据操作语句都可以用于执行部分，PL/SQL 块不能再屏幕上显示 SELECT 语句的输出。SELECT 语句必须包括一个 INTO 子串或者是游标的一部分，执行部分使用的变量和常量必须首先在声明部分声明，执行部分必须至少包括一条可执行语句，NULL 是一条合法的可执行语句，事物控制语句 COMMIT 和 ROLLBACK 可以在执行部分使用，数据定义语言(Data Definition language)不能在执行部分中使用，DDL 语句与 EXECUTE IMMEDIATE 一起使用或者是 DBMS_SQL 调用。

执行一个 PL/SQL 块

SQL*PLUS 中匿名的 PL/SQL 块的执行是在 PL/SQL 块后输入/来执行，如下面的例子所示：

```
declare
   v_comm_percent constant number:=10;
begin
   update emp
   set comm=sal*v_comm_percent
   where deptno=10;
   end
SQL> /
PL/SQL procedure successfully completed.

SQL>
```

命名的程序与匿名程序的执行不同，执行命名的程序块必须使用 execute 关键字：

```
create or replace procedure update_commission
   (v_dept in number,v_pervent in number default 10) is
begin
   update emp
   set comm=sal*v_percent
   where deptno=v_dept;
end

SQL>/

Procedure created

SQL>execute update_commission(10,15);

PL/SQL procedure successfully completed.

SQL>
```

如果在另一个命名程序块或匿名程序块中执行这个程序，那么就不需要 EXECUTE 关进字。

```
declare
   v_dept number;
begin
   select a.deptno
   into v_dept
   from emp a
   where job='PRESIDENT'
   update_commission(v_dept);
end
SQL>/
   PL/SQL procedure successfully completed
SQL>
```

PL/SQL 语言基础

Oracle PL/SQL 语言基础(3)

控制结构

控制结构控制 PL/SQL 程序流程的代码行,PL/SQL 支持条件控制和循环控制结构。

语法和用途

IF..THEN

语法：

```
IF condition THEN
   Statements 1;
   Statements 2;
   ....
END IF
```

IF 语句判断条件 condition 是否为 TRUE，如果是，则执行 THEN 后面的语句，如果 condition 为 false 或 NULL 则跳过 THEN 到 END IF 之间的语句，执行 END IF 后面的语句。

IF..THEN...ELSE

语法:

```
IF condition THEN
   Statements 1;
   Statements 2;
   ....
ELSE
   Statements 1;
   Statements 2;
   ....
END IF
```

如果条件 condition 为 TRUE,则执行 THEN 到 ELSE 之间的语句，否则执行 ELSE 到 END IF 之间的语句。

IF 可以嵌套，可以在 IF 或 IF ..ELSE 语句中使用 IF 或 IF..ELSE 语句。

```
if (a>b) and (a>c) then
      g:=a;
```

```
else
      g:=b;
      if c>g then
         g:=c;
      end if
end if
```

IF..THEN..ELSIF

语法：

```
IF condition1 THEN
   statement1;
ELSIF condition2 THEN
   statement2;
ELSIF condition3 THEN
   statement3;
ELSE
   statement4;
END IF;
   statement5;
```

如果条件 condition1 为 TRUE 则执行 statement1,然后执行 statement5,否则判断 condition2 是否为 TRUE,若为 TRUE 则执行 statement2,然后执行 statement5,对于 condition3 也是相同的，如果 condition1，condition2，condition3 都不成立，那么将执行 statement4,然后执行 statement5。

循环控制

循环控制的基本形式是 LOOP 语句，LOOP 和 END LOOP 之间的语句将无限次的执行。LOOP 语句的语法如下：

```
LOOP
   statements;
END LOOP
```

LOOP 和 END LOOP 之间的语句无限次的执行显然是不行的，那么在使用 LOOP 语句时必须使用 EXIT 语句，强制循环结束，例如：

```
X:=100;
LOOP
  X:=X+10;
  IF X>1000 THEN
    EXIT;
  END IF
END LOOP;
Y:=X;
```

此时 Y 的值是 1010.

EXIT WHEN 语句将结束循环，如果条件为 TRUE，则结束循环。

```
X:=100;
LOOP
X:=X+10;
EXIT WHEN X>1000;
X:=X+10;
END LOOP;
Y:=X;
```

WHILE..LOOP

WHILE..LOOP 有一个条件与循环相联系，如果条件为 TRUE,则执行循环体内的语句，如果结果为 FALSE,则结束循环。

```
X:=100;
WHILE X<=1000 LOOP
   X:=X+10;
END LOOP;
Y=X;
```

FOR...LOOP

语法：

```
FOR counter IN [REVERSE] start_range....end_range LOOP
statements;
END LOOP;
```

LOOP 和 WHILE 循环的循环次数都是不确定的，FOR 循环的循环次数是固定的，counter 是一个隐式声明的变量，他的初始值是 start_range,第二个值是 start_range+1,直到 end_range,如果 start_range 等于 end _range,那么循环将执行一次。如果使用了 REVERSE 关键字，那么范围将是一个降序。

```
X:=100;
FOR v_counter in 1..10 loop
x:=x+10;

end loop
y:=x;
```

如果要退出 for 循环可以使用 EXIT 语句。

标签

用户可以使用标签使程序获得更好的可读性。程序块或循环都可以被标记。标签的形式

是<>。

标记程序块

<>
[DECLARE]
... ...
BEGIN
.....
[EXCEPTION]
.....
END label_name

标记循环

<>
LOOP
.....
<>
loop
.....
<>
loop
....

EXIT outer_loop WHEN v_condition=0;
end loop innermost_loop;
.....
END LOOP inner_loop;
END LOOP outer_loop;

GOTO 语句

语法：

GOTO LABEL;

执行 GOTO 语句时，控制会立即转到由标签标记的语句。PL/SQL 中对 GOTO 语句有一些限制，对于块、循环、IF 语句而言，从外层跳转到内层是非法的。

```
X ：=100;
FOR V_COUNTER IN 1..10 LOOP
  IF V_COUNTER =4 THEN
    GOTO end_of_loop
  END IF
  X:=X+10;
  <>
  NULL
END LOOP

Y:=X;
```

注意：NULL 是一个合法的可执行语句。

嵌套

程序块的内部可以有另一个程序块这种情况称为嵌套。嵌套要注意的是变量，定义在最外部程序块中的变量可以在所有子块中使用，如果在子块中定义了与外部程序块变量相同的变量名，在执行子块时将使用子块中定义的变量。子块中定义的变量不能被父块引用。同样 GOTO 语句不能由父块跳转道子块中，反之则是合法的。

```
《OUTER BLOCK》
DECLARE
  A_NUMBER INTEGER;
  B_NUMBER INTEGER;
BEGIN
  --A_NUMBER and B_NUMBER are available here
  <>
  DECLARE
  C_NUMBER INTEGER
  B_NUMBER NUMBER(20)
```

```
BEGIN
   C_NUMBER:=A_NUMBER;
   C_NUMBER=OUTER_BLOCK.B_NUMBER;
END SUB_BLOCK;
END OUT_BLOCK;
```

小结

我们在这篇文章中介绍了 PL/SQL 的基础语法以及如何使用 PL/SQL 语言设计和运行 PL/SQL 程序块，并将 PL/SQL 程序整合到 Oracle 服务器中,虽然 PL/SQL 程序作为功能块嵌入 Oracle 数据库中，但 PL/SQL 与 ORACLE 数据库的紧密结合使得越来越多的 Oracle 数据库管理员和开发人员开始使用 PL/SQL。

复合数据类型

复合数据类型(1)

PL/SQL 有两种复合数据结构：记录和集合。记录由不同的域组成，集合由不同的元素组成。在本文中我们将讨论记录和集合的类型、怎样定义和使用记录和集合。

PL/SQL 记录

记录是 PL/SQL 的一种复合数据结构，scalar 数据类型和其他数据类型只是简单的在包一级进行预定义，但复合数据类型在使用前必须被定义，记录之所以被称为复合数据类型是因为他由域这种由数据元素的逻辑组所组成。域可以是 scalar 数据类型或其他记录类型，

它与 c 语言中的结构相似，记录也可以看成表中的数据行，域则相当于表中的列，在表和虚拟表（视图或查询）中非常容易定义和使用，行或记录中的每一列或域都可以被引用或单独赋值，也可以通过一个单独的语句引用记录所有的域。在存储过程或函数中记录也可能有参数。

创建记录

在 PL/SQL 中有两种定义方式：显式定义和隐式定义。一旦记录被定义后，声明或创建定义类型的记录变量，然后才是使用该变量。隐式声明是在基于表的结构或查询上使用 %TYPE 属性，隐式声明是一个更强有力的工具，这是因为这种数据变量是动态创建的。

显式定义记录

显式定义记录是在 PL/SQL 程序块中创建记录变量之前在声明部分定义。使用 type 命令定义记录，然后在创建该记录的变量。语法如下：

```
TYPE record_type IS RECORD (field_definition_list);
```

field_definition_list 是由逗号分隔的列表。

域定义的语法如下：

```
field_name data_type_and_size [NOT NULL][{:=|DEFAULT} default_value]
```

域名必须服从与表或列的命名规则相同的命名规则。下面我们看一个例子：

```
DELCARE
TYPE stock_quote_rec IS RECORD
```

```
(symbol stock.symbol%TYPE
,bid NUMBER(10,4)
,ask NUMBER(10,4)
,volume NUMBER NOT NULL:=0
,exchange VARCHAR2(6) DEFAULT 'NASDAQ'
);

real_time_quote stock_quote_rec;
variable
```

域定义时的%TYPE 属性用于引用数据库中的表或视图的数据类型和大小,而在此之前程序不知道类型和大小。在上面的例子中记录域在编译时将被定义为与列 SYMBOL 相同的数据类型和大小,当代码中要使用来自数据库中的数据时，在变量或域定义中最好使用 %TYPE 来定义。

隐式定义记录

隐式定义记录中，我们不用描述记录的每一个域。这是因为我们不需要定义记录的结构，不需要使用 TYPE 语句，相反在声明记录变量时使用%ROWTYPE 命令定义与数据库表，视图，游标有相同结构的记录，与 TYPE 命令相同的是它是一种定义获得数据库数据记录的好方法。

```
DECLARE

accounter_info accounts%ROWTYPR;

CURSOR xactions_cur(acct_no IN VARCHAR2) IS
SELECT action,timestamp,holding
FROM portfolios
WHERE account_nbr='acct_no'
;
xaction_info xactions_cur%ROWTYPE;
variable
```

有一些 PL/SQL 指令在使用隐式定义记录时没有使用%ROWTYPE 属性，比如游标 FOR 循环或触发器中的:old 和:new 记录。

```
DELCARE

CURSOR xaction_cur IS
SELECT action,timeamp,holding
FROM portfolios
WHERE account_nbr='37'
;

BEGIN
FOR xaction_rec in xactions_cur
LOOP
IF xactions_rec.holding='ORCL'
THEN
notify_shareholder;
END IF;
END LOOP;
```

复合数据类型

复合数据类型(2)

使用记录

用户可以给记录赋值、将值传递给其他程序。记录作为一种复合数据结构意味作他有两个层次可用。用户可以引用整个记录，使用 select into 或 fetch 转移所有域，也可以将整个记录传递给一个程序或将所有域的值赋给另一个记录。在更低的层次，用户可以处理记录内单独的域，用户可以给单独的域赋值或者在单独的域上运行布尔表达式，也可以将一个或更多的域传递给另一个程序。

引用记录

记录由域组成，访问记录中的域使用点（.）符号。我们使用上面的例子看看

```
DELCARE
TYPE stock_quote_rec IS RECORD
(symbol stock.symbol%TYPE
,bid NUMBER(10,4)
,ask NUMBER(10,4)
,volume NUMBER NOT NULL:=0
,exchange VARCHAR2(6) DEFAULT 'NASDAQ'
);

TYPE detailed_quote_rec IS RECORD
(quote stock_quote_rec
,timestamp date
,bid_size NUMBER
,ask.size NUMBER
,last_tick VARCHAR2(4)
);

real_time_detail detail_quote_rec;

BEGIN

real_time_detail.bid_size:=1000;
real_time_detail.quote.volume:=156700;
log_quote(real_time_detail.quote);
```

给记录赋值

给记录或记录中的域赋值的方法有几种，可以使用 SELECT INTO 或 FETCH 给整个记录或单独的域赋值， 可以将整个记录的值赋给其他记录，也可以通过给每个域赋值来得到记录，以下我们通过实例讲解每一种赋值方法。

1、使用 SELECT INTO

使用 SELECT INTO 给记录赋值要将记录或域放在 INTO 子串中，INTO 子串中的变量与 SELECT 中列的位置相对应。

例：

```
DECLARE

stock_info1 stocks%ROWTYPE;
stock_info2 stocks%ROWTYPE;

BEGIN

SELECT symbol,exchange
INTO stock_info1.symbol,stock_info1.exchange
FROM stocks
WHERE symbol='ORCL';

SELECT * INTO stock_info2 FROM stocks
WHERE symbol='ORCL';
```

2、使用 FETCH

如果 SQL 语句返回多行数据或者希望使用带参数的游标，那么就要使用游标，这种情况下使用 FETCH 语句代替 INSTEAD INTO 是一个更简单、更有效率的方法，但在安全性较高的包中 FETCH 的语法如下：

```
FETCH cursor_name INTO variable;
```

我们改写上面的例子：

```
DECLARE
CURSOR stock_cur(symbol_in VARCHAR2) IS
SELECT symbol,exchange,begin_date
FROM stock
WHERE symbol=UPPER(symbol_in);

stock_info stock_cur%ROWTYPE

BEGIN
OPEN stock_cur('ORCL');
FETCH stock_cur INTO stock_info;
```

使用赋值语句将整个记录复制给另一个记录是一项非常有用的技术，不过记录必须精确

地被声明为相同的类型，不能是基于两个不同的 TYPE 语句来获得相同的结构。

例：

```
DECLARE

TYPE stock_quote_rec IS RECORD
(symbol stocks.symbol%TYPE
,bid NUMBER(10,4)
,ask number(10,4)
,volume NUMBER
);

TYPE stock_quote_too IS RECORD
(symbol stocks.symbol%TYPE
,bid NUMBER(10,4)
,ask number(10,4)
,volume NUMBER
);
--这两个记录看上去是一样的，但实际上是不一样的
stock_one stocks_quote_rec;
stock_two stocks_quote_rec;
--这两个域有相同的数据类型和大小

stock_also stock_rec_too；--与 stock_quote_rec 是不同的数据类型
BEGIN
stock_one.symbol:='orcl';
stock_one.volume:=1234500;
stock_two：=stock_one;--正确
syock_also：=stock_one;--错误，数据类型错误

stock_also.symbol:=stock_one.symbol;
stock_also.volume:=stock_one.volume;
```

记录不能用于 INSERT 语句和将记录直接用于比较，下面两种情况是错误的：

```
INSERT INTO stocks VALUES (stock_record);
```

和

```
IF stock_rec1>stock_rec2 THEN
```

要特别注意考试中试题中有可能用%ROWTYPE 来欺骗你，但这是错误的，记住这一点。还有可能会出现用记录排序的情况，ORACLE 不支持记录之间的直接比较。对于记录比较，可以采用下面的两个选择：

．设计一个函数，该函数返回 scalar 数据类型，使用这个函数比较记录，如

IF sort_rec(stock_one)>sort_rec(stock_two) THEN

．可以使用数据库对象，数据库对象可以使用 order 或 map 方法定义，允许 oracle 对复合数据类型进行比较。关于数据库对象的讨论已经超越了本文的范围，要详细了解数据库对象，可以查阅 oracle 手册。

复合数据类型

复合数据类型(3)

PL/SQL 集合

集合与其他语言中的数组相似，在 ORACLE7.3 及以前的版本中只有一种集合称为 PL/SQL 表，这种类型的集合依然保留，就是索引（INDEX_BY）表，与记录相似，集合在定义的时候必须使用 TYPE 语句，然后才是创建和使用这种类型的变量。

集合的类型

PL/SQL 有三种类型的集合

. Index_by 表

. 嵌套表

. VARRAY

这三种类型的集合之间由许多差异，包括数据绑定、稀疏性(sparsity)、数据库中的存储能力都不相同。绑定涉及到集合中元素数量的限制，VARRAY 集合中的元素的数量是有限，Index_by 和嵌套表则是没有限制的。稀疏性描述了集合的下标是否有间隔，Index_by 表总是稀疏的，如果元素被删除了嵌套表可以是稀疏的，但 VARRAY 类型的集合则是紧密的，它的下标之间没有间隔。

Index_by 表不能存储在数据库中，但嵌套表和 VARRAY 可以被存储在数据库中。

虽然这三种类型的集合有很多不同之处，但他们也由很多相似的地方：

. 都是一维的类似数组的结构

. 都有内建的方法

. 访问由点分隔

Index_by 表

Index_by 表集合的定义语法如下：

```
TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX
BY BINARY_INTERGET;
```

这里面重要的关键字是 INDEX BY BINARY_INTERGET，没有这个关键字，那么集合将是一个嵌套表，element_type 可以是任何合法的 PL/SQL 数据类型，包括：PLS/INTEGER、SIGNTYPE、和 BOOLEAN。其他的集合类型对数据库的数据类型都有限制，但 Index_by 表不能存储在数据库中，所以没有这些限制。

一旦定义了 index_by 表，就可以向创建其他变量那样创建 index_by 表的变量：

```
DECLARE
TYPE symbol_tab_typ IS TABLE OF VARCHAR2(5) INDEX BY BINARY_INTEGER;
symbol_tab symbol_tab_typ;
BEGIN
```

嵌套表

嵌套表非常类似于 Index_by 表，创建的语法也非常相似。使用 TYPE 语句，只是没有 INDEX BY BINARY_INTEGER 子串。

TYPE type_name IS TABLE OF element_type [NOT NULL]

NOT NULL 选项要求集合所有的元素都要有值，element_type 可以是一个记录，但是这个记录只能使用标量数据类型字段以及只用于数据库的数据类型（不能是 PLS_INTEGER,BOOLEAN 或 SIGNTYPE）。

嵌套表和 VARRAY 都能作为列存储在数据库表中，所以集合自身而不是单个的元素可以为 NULL,ORACLE 称这种整个集合为 NULL 的为"自动设置为 NULL(atomically NULL)"

以区别元素为 NULL 的情况。当集合为 NULL 时，即使不会产生异常，用户也不能引用集合中的元素。用户可以使用 IS NULL 操作符检测集合是否为 NULL。

存储在一个数据库中的嵌套表并不与表中的其它数据存放在同一个数据块中，它们实际上被存放在第二个表中。正如没有 order by 子句 select 语句不能保证返回任何有顺序的数据，从数据库中取回的嵌套表也不保证元素的顺序。由于集合数据是离线存储的，对于大型集合嵌套表是一个不错的选择。

VARRAY

VARRAY 或数据变量都有元素的限制。想起他集合一样 VARRAY 定义仍然使用 TYPE 语句，但关键字 VARRAY 或 VARRYING ARRAY 告诉 ORACLE 这是一个 VARRAY 集合。

```
TYPE type_name IS [VARRAY|VARYING ARRAY] (max_size) OF
element_type [NOT NULL]
```

max_size 是一个整数，用于标示 VARRAY 集合拥有的最多元素数目。VARRAY 集合的元素数量可以低于 max_size,但不能超过 max_size。element_type 是一维元素的数据类型，如果 element_type 是记录，那么这个记录只能使用标量数据字段（与嵌套标相似）。NOT NULL 子串表示集合中的每一个元素都必须有值。

与嵌套表相似，VARRAY 能够自动为 NULL,可以使用 IS NULL 操作符进行检测。与嵌套表不同的是，当 VARRAY 存储在数据库中时与表中的其他数据存放在同一个数据块中。正象列的排序保存在表的 SELECT*中一样元素的顺序保存在 VARRAY 中。同样由于集合是在线存储的，VARRAY 很适合于小型集合。

复合数据类型(4)

使用集合

象记录一样，集合可以在两个层面上使用：

. 操作整个集合

. 访问集合中的单个元素

第一种情况使用集合名，第二种情况使用下标：

collection(subscript)

index_by 表的下标是两为的整数，可以为正也可以为负，范围是：-2147483647--2147483647。嵌套表和 VARRAY 表示元素在集合中的位置，用户很难灵活设计下标，这是因为：

. 嵌套表开始是紧密的（相对于疏松）

. VARRAY 始终保持紧密

. 这两种集合的下标都由 1 开始

初始化、删除、引用集合

使用集合之前必须要初始化，对于 Index_by 表初始化是自动进行的，但是对于嵌套表和 VARRAY 就必须使用内建的构造函数。如果重新调用，嵌套表和 VARRAY 自动置 NULL，这不只是元素置 NULL，而是整个集合置 NULL。给集合内的元素赋值需要使用下标符号。将一个集合的值赋给另一个集合，只需要简单的使用赋值操作符。

Index_by 集合初始化是最简单的，只要涉及其中的一个元素集合就被初始化了。

例：

```
DECLARE

TYPE symbol_tab_typ IS TABLE OF VARCHAR2(5) INDEX BY BINARY_INTEGER;
TYPE account_tab_typ IS TABLE OF account%ROWTYPE INDEX BY BINARY_INTEGER;
symbol_tab symbol_tab_typ;
account_tab account_tab_typ;
new_acct_tab account_tab_typ;

BEGIN
--初始化集合元素 147 和-3
SELECT * INTO account_tab(147)
FROM accounts WHERE account_nbr=147;

SELECT * INTO account_tab(-3)
FROM accounts WHERE account_nbr=3003;

IF account_tab(147).balance<500 THEN
chang_maintenance_fee(147);
END IF

new_acct_tab:=account_tab;
symbol_tab(1):="ORCL";
symbol_tab(2):="CSCO";
symbol_tab(3):="SUNM";

publish_portfolio(symbol_tab);
```

嵌套表和 VARRAY 由构造函数初始化，构造函数和集合的名字相同，同时有一组参数，每个参数对应一个元素，如果参数为 NULL,那么对应的元素就被初始化为 NULL,如果创建了元素，但没有填充数据，那么元素将保持 null 值，可以被引用，但不能保持数据。如果元素没有初始化，那么就不能引用该元素。

例：

```
DECLARE

TYPE stock_list IS TABLE OF stock.symbol%TYPE;
TYPE top10_list IS VARRAY (10) OF stocks.symbol%TYPE;
```

```
biotech_stocks stock_list;
tech_10 top10_list;

BEGIN
--非法，集合未初始化。
biotech_stocks(1):='AMGN';
IF biotech_stocks IS NULL THEN
--初始化集合
biotech_stocks：=('AMGN','BGEN',IMCL','GERN',CRA');
END IF;
tech_10:=top10_list('ORCL',CSCO','MSFT','INTC','SUNW','IBM',NULL,NULL);
IF tech_10(7) IS NULL THEN
tech_10(7):='CPQ';
END
tech_10(8):='DELL';
```

在这个例子中，嵌套表 BIOTECH_STOCKS 初始化有 5 个元素，VARRAY tech_10 集合最多能有 10 个元素，但构造函数只创建了 8 个元素，其中还有两个元素是 NULL 值，并程序中给他们赋值。

初始化基于记录的集合，就必须将记录传递给构造函数，注意不能只是简单的将记录的域传递给构造函数。

例：

```
DECLARE

TYPE stock_quote_rec IS RECORD
(symbol stock.symbol%TYPE
,bid NUMBER(10,4)
,ask NUMBER(10,4)
,volume NUMBER NOT NULL:=0
);
TYPE stock_tab_typ IS TABLE OF stock_quote_rec;
quote_list stock_tab_typ;
single_quote stock_quote_rec;

BEGIN
single_quote.symbol:='OPCL';
single_quote.bid:=100;
single_quote.ask:=101;
single_quote.volume:=25000;
```

```
--合法
quote_list:=stock_tab_typ(single_quote);
--不合法
quote_list:=stock_tab_typ('CSCO',75,76,3210000);
DBMS_OUTPUT.LINE(quote_list(1).bid);
```

复合数据类型

复合数据类型(5)

集合的方法

除了构造函数外,集合还有很多内建函数，这些函数称为方法。调用方法的语法如下：

collection.method

下表中列出 oracle 中集合的方法

方法	描述	使用限制
COUNT	返回集合中元素的个数	
DELETE	删除集合中所有元素	
DELETE()	删除元素下标为 x 的元素，如果 x 为 null,则集合保持不变	对 VARRAY 非法
DELETE(,)	删除元素下标从 X 到 Y 的元素，如果 X>Y 集合保持不变	对 VARRAY 非法
EXIST()	如果集合元素 x 已经初始化，则返回 TRUE，否则返回 FALSE	
EXTEND	在集合末尾添加一个元素	对 Index_by 非法
EXTEND()	在集合末尾添加 x 个元素	对 Index_by 非法
EXTEND(,)	在集合末尾添加元素 n 的 x 个副本	对 Index_by 非法
FIRST	返回集合中的第一个元素的下标号，对于 VARRAY 集合始终返回 1。	
LAST	返回集合中最后一个元素的下标号，对于 VARRAY 返回值始终等于 COUNT.	
LIMIT	返回 VARRY 集合的最大的元素个数，对于嵌套表和对于嵌套表和 Index_by 为 null	Index_by 集合无用
NEXT()	返回在元素 x 之后及紧挨着它的元素的值，如果该元素是最后一个元素，则返回 null.	

PRIOR()	返回集合中在元素 x 之前紧挨着它的元素的值，如果该元素是第一个元素，则返回 null。	
TRI M	从集合末端开始删除一个元素	对于 index_by 不合法
TRIM()	从集合末端开始删除 x 个元素	对 index_by 不合法

关于集合之间的比较

集合不能直接用于比较，要比较两个集合，可以设计一个函数，该函数返回一个标量数据类型。

```
IF stock_list1>stock_list2 ----非法
IF sort_collection(stock_list1)>sort_collection(stock_list2) THEN --合法
```

但可以比较在集合内的两个元素。

单行函数和组函数

单行函数和组函数详解(1)

函数是一种有零个或多个参数并且有一个返回值的程序。在 SQL 中 Oracle 内建了一系列函数，这些函数都可被称为 SQL 或 PL/SQL 语句，函数主要分为两大类：

单行函数

组函数

本文将讨论如何利用单行函数以及使用规则。

SQL 中的单行函数

SQL 和 PL/SQL 中自带很多类型的函数，有字符、数字、日期、转换、和混合型等多种函数用于处理单行数据，因此这些都可被统称为单行函数。这些函数均可用于 SELECT,WHERE、ORDER BY 等子句中，例如下面的例子中就包含了 TO_CHAR,UPPER,SOUNDEX 等单行函数。

```
SELECT ename,TO_CHAR(hiredate,'day,DD-Mon-YYYY')
FROM emp
Where UPPER(ename) Like 'AL%'
ORDER BY SOUNDEX(ename)
```

单行函数也可以在其他语句中使用，如 update 的 SET 子句，INSERT 的 VALUES 子句，DELET 的 WHERE 子句,认证考试特别注意在 SELECT 语句中使用这些函数，所以我们的注意力也集中在 SELECT 语句中。

NULL 和单行函数

在如何理解 NULL 上开始是很困难的，就算是一个很有经验的人依然对此感到困惑。NULL 值表示一个未知数据或者一个空值，算术操作符的任何一个操作数为 NULL 值，结果均为提个 NULL 值,这个规则也适合很多函数，只有 CONCAT,DECODE,DUMP,NVL,REPLACE 在调用了 NULL 参数时能够返回非 NULL 值。在这些中 NVL 函数时最重要的，因为他能直接处理 NULL 值，NVL 有两个参数：NVL(x1,x2),x1 和 x2 都式表达式，当 x1 为 null 时返回 X2,否则返回 x1。

下面我们看看 emp 数据表它包含了薪水、奖金两项，需要计算总的补偿

column name emp_id salary bonus

key type pk
nulls/unique nn,u nn
fk table
datatype number number number
length 11.2 11.2

不是简单的将薪水和奖金加起来就可以了，如果某一行是 null 值那么结果就将是 null，比如下面的例子：

update emp
set salary=(salary+bonus)*1.1

这个语句中，雇员的工资和奖金都将更新为一个新的值，但是如果没有奖金，即 salary + null,那么就会得出错误的结论，这个时候就要使用 nvl 函数来排除 null 值的影响。所以正确的语句是：

update emp
set salary=(salary+nvl(bonus,0)*1.1

单行函数和组函数

单行函数和组函数详解(2)

单行字符串函数

单行字符串函数用于操作字符串数据，他们大多数有一个或多个参数，其中绝大多数返回字符串

ASCII()

c1 是一字符串，返回 c1 第一个字母的 ASCII 码，他的逆函数是 CHR()

SELECT ASCII('A') BIG_A,ASCII('z') BIG_z FROM emp

BIG_A BIG_z
65 122

CHR(＜i＞)[NCHAR_CS]

i 是一个数字，函数返回十进制表示的字符

select CHR(65),CHR(122),CHR(223) FROM emp

CHR65 CHR122 CHR223
A z B

CONCAT(,)

c1,c2 均为字符串，函数将 c2 连接到 c1 的后面，如果 c1 为 null,将返回 c2.如果 c2 为 null,则返回 c1，如果 c1、c2 都为 null，则返回 null。他和操作符||返回的结果相同

select concat('slobo ','Svoboda') username from dual

username

slobo Syoboda

INITCAP()

c1 为一字符串。函数将每个单词的第一个字母大写其它字母小写返回。单词由空格，控制字符，标点符号限制。

select INITCAP('veni,vedi,vici') Ceasar from dual

Ceasar

Veni,Vedi,Vici

INSTR(,[,＜i＞[,]])

c1,c2 均为字符串，i,j 为整数。函数返回 c2 在 c1 中第 j 次出现的位置，搜索从 c1 的第 i 个字符开始。当没有发现需要的字符时返回 0,如果 i 为负数，那么搜索将从右到左进行，但是位置的计算还是从左到右，i 和 j 的缺省值为 1.

select INSTR('Mississippi','i',3,3) from dual

INSTR('MISSISSIPPI','I',3,3)

11

select INSTR('Mississippi','i',-2,3) from dual

INSTR('MISSISSIPPI','I',3,3)

2

INSTRB(,[,i[,j])

与 INSTR（）函数一样，只是他返回的是字节，对于单字节 INSTRB()等于 INSTR()

LENGTH()

c1 为字符串，返回 c1 的长度，如果 c1 为 null，那么将返回 null 值。

select LENGTH('Ipso Facto') ergo from dual

ergo

10

LENGTHb()

与 LENGTH()一样，返回字节。

lower()

返回 c 的小写字符，经常出现在 where 子串中

```
select LOWER(colorname) from itemdetail WHERE LOWER(colorname) LIKE '%white%'

COLORNAME

Winterwhite
```

LPAD(,＜i＞[,])

c1,c2 均为字符串，i 为整数。在 c1 的左侧用 c2 字符串补足致长度 i,可多次重复，如果 i 小于 c1 的长度，那么只返回 i 那么长的 c1 字符，其他的将被截去。c2 的缺省值为单空格，参见 RPAD。

```
select LPAD(answer,7,'') padded,answer unpadded from question;

PADDED UNPADDED

Yes Yes
NO NO
Maybe maybe
```

LTRIM(,)

把 c1 中最左边的字符去掉，使其第一个字符不在 c2 中，如果没有 c2，那么 c1 就不会改变。

```
select LTRIM('Mississippi','Mis') from dual

LTR

ppi
```

RPAD(,＜i＞[,])

在 c1 的右侧用 c2 字符串补足致长度 i,可多次重复，如果 i 小于 c1 的长度，那么只返回 i 那么长的 c1 字符，其他的将被截去。c2 的缺省值为单空格,其他与 LPAD 相似

RTRIM(,)

把 c1 中最右边的字符去掉，使其第后一个字符不在 c2 中，如果没有 c2，那么 c1 就不会改变。

REPLACE(,[,])

c1,c2,c3 都是字符串，函数用 c3 代替出现在 c1 中的 c2 后返回。

select REPLACE('uptown','up','down') from dual

REPLACE

downtown

STBSTR(,＜i＞[,])

c1 为一字符串，i,j 为整数，从 c1 的第 i 位开始返回长度为 j 的子字符串，如果 j 为空，则直到串的尾部。

select SUBSTR('Message',1,4) from dual

SUBS

Mess

SUBSTRB(,＜i＞[,])

与 SUBSTR 大致相同，只是 I,J 是以字节计算。

SOUNDEX()

返回与 c1 发音相似的词

select SOUNDEX('dawes') Dawes SOUNDEX('daws') Daws, SOUNDEX('dawson') from dual

Dawes Daws Dawson

D200 D200 D250

TRANSLATE(,,)

将 c1 中与 c2 相同的字符以 c3 代替

select TRANSLATE('fumble','uf','ar') test from dual

TEXT

ramble

TRIM([[]] from c3)

将 c3 串中的第一个，最后一个，或者都删除。

select TRIM(' space padded ') trim from dual

TRIM

space padded

UPPER()

返回 c1 的大写，常出现 where 子串中

select name from dual where UPPER(name) LIKE 'KI%'

NAME

KING

单行函数和组函数

单行函数和组函数详解(3)

单行数字函数

单行数字函数操作数字数据，执行数学和算术运算。所有函数都有数字参数并返回数字值。所有三角函数的操作数和值都是弧度而不是角度，oracle 没有提供内建的弧度和角度的转换函数。

ABS()
返回 n 的绝对值

ACOS()
反余玄函数，返回-1 到 1 之间的数。n 表示弧度

select ACOS(-1) pi,ACOS(1) ZERO FROM dual

PI ZERO

3.14159265 0

ASIN()
反正玄函数，返回-1 到 1，n 表示弧度

ATAN()
反正切函数，返回 n 的反正切值，n 表示弧度。

CEIL()
返回大于或等于 n 的最小整数。

COS()

返回 n 的余玄值，n 为弧度

COSH()

返回 n 的双曲余玄值，n 为数字。

select COSH(<1.4>) FROM dual

COSH(1.4)

2.15089847

EXP()

返回 e 的 n 次幂，e=2.71828183.

FLOOR()

返回小于等于 N 的最大整数。

LN()

返回 N 的自然对数，N 必须大于 0

LOG(,)

返回以 n1 为底 n2 的对数

MOD()

返回 n1 除以 n2 的余数，

POWER(,)

返回 n1 的 n2 次方

ROUND(,)

返回舍入小数点右边 n2 位的 n1 的值，n2 的缺省值为 0，这回将小数点最接近的整数，如果 n2 为负数就舍入到小数点左边相应的位上，n2 必须是整数。

select ROUND(12345,-2),ROUND(12345.54321,2) FROM dual

ROUND(12345,-2) ROUND(12345.54321,2)

12300 12345.54

SIGN()

如果 n 为负数，返回-1,如果 n 为正数，返回 1，如果 n=0 返回 0.

SIN（)

返回 n 的正玄值,n 为弧度。

SINH()

返回 n 的双曲正玄值,n 为弧度。

SQRT()

返回 n 的平方根,n 为弧度

TAN（)

返回 n 的正切值,n 为弧度

TANH()

返回 n 的双曲正切值,n 为弧度

TRUNC(,)

返回截尾到 n2 位小数的 n1 的值，n2 缺省设置为 0，当 n2 为缺省设置时会将 n1 截尾为整数，如果 n2 为负值，就截尾在小数点左边相应的位上。

单行日期函数

单行日期函数操作 DATA 数据类型，绝大多数都有 DATA 数据类型的参数，绝大多数返回的也是 DATA 数据类型的值。

ADD_MONTHS(,＜i＞)

返回日期 d 加上 i 个月后的结果。i 可以使任意整数。如果 i 是一个小数，那么数据库将隐式的他转换成整数，将会截去小数点后面的部分。

LAST_DAY()

函数返回包含日期 d 的月份的最后一天

MONTHS_BETWEEN(,)

返回 d1 和 d2 之间月的数目,如果 d1 和 d2 的日的日期都相同，或者都使该月的最后一天，那么将返回一个整数，否则会返回的结果将包含一个分数。

NEW_TIME(,,)

d1 是一个日期数据类型，当时区 tz1 中的日期和时间是 d 时，返回时区 tz2 中的日期和时间。tz1 和 tz2 时字符串。

NEXT_DAY(,)

返回日期 d 后由 dow 给出的条件的第一天，dow 使用当前会话中给出的语言指定了一周中的某一天，返回的时间分量与 d 的时间分量相同。

select NEXT_DAY('01-Jan-2000','Monday') "1st Monday",NEXT_DAY('01-Nov-2004','Tuesday')+7 "2nd Tuesday") from dual;

1st Monday 2nd Tuesday

03-Jan-2000 09-Nov-2004

ROUND([,])

将日期 d 按照 fmt 指定的格式舍入，fmt 为字符串。

SYADATE

函数没有参数，返回当前日期和时间。

TRUNC([,])

返回由 fmt 指定的单位的日期 d.

单行函数和组函数

单行函数和组函数详解(4)

单行转换函数

单行转换函数用于操作多数据类型，在数据类型之间进行转换。

CHARTORWID()

c 使一个字符串，函数将 c 转换为 RWID 数据类型。

```
SELECT test_id from test_case where rowid=CHARTORWID('AAAA0SAACAAAALiAAA')
```

CONVERT(,[,])

c 尾字符串，dset、sset 是两个字符集，函数将字符串 c 由 sset 字符集转换为 dset 字符集，sset 的缺省设置为数据库的字符集。

HEXTORAW()

x 为 16 进制的字符串，函数将 16 进制的 x 转换为 RAW 数据类型。

RAWTOHEX()

x 是 RAW 数据类型字符串，函数将 RAW 数据类转换为 16 进制的数据类型。

ROWIDTOCHAR()

函数将 ROWID 数据类型转换为 CHAR 数据类型。

TO_CHAR([[,)

x 是一个 data 或 number 数据类型，函数将 x 转换成 fmt 指定格式的 char 数据类型，如果 x 为日期 nlsparm=NLS_DATE_LANGUAGE 控制返回的月份和日份所使用的语言。如果 x 为数字 nlsparm=NLS_NUMERIC_CHARACTERS 用来指定小数位和千分位的分隔符，以及货币符号。

```
NLS_NUMERIC_CHARACTERS ="dg", NLS_CURRENCY="string"
```

TO_DATE([,[,）

c 表示字符串，fmt 表示一种特殊格式的字符串。返回按照 fmt 格式显示的 c,nlsparm 表示使用的语言。函数将字符串 c 转换成 date 数据类型。

TO_MULTI_BYTE()

c 表示一个字符串，函数将 c 的担子截字符转换成多字节字符。

TO_NUMBER([,[,)

c 表示字符串，fmt 表示一个特殊格式的字符串，函数返回值按照 fmt 指定的格式显示。nlsparm 表示语言，函数将返回 c 代表的数字。

TO_SINGLE_BYTE()

将字符串 c 中得多字节字符转化成等价的单字节字符。该函数仅当数据库字符集同时包含单字节和多字节字符时才使用

其它单行函数

BFILENAME(

,)

dir 是一个 directory 类型的对象，file 为一文件名。函数返回一个空的 BFILE 位置值指示符，函数用于初始化 BFILE 变量或者是 BFILE 列。

DECODE(,,[,,,[])

x 是一个表达式，m1 是一个匹配表达式，x 与 m1 比较，如果 m1 等于 x，那么返回 r1,否则,x 与 m2 比较，依次类推 m3,m4,m5....直到有返回结果。

DUMP(,[,[,[,]]])

x 是一个表达式或字符，fmt 表示 8 进制、10 进制、16 进制、或则单字符。函数返回包含了有关 x 的内部表示信息的 VARCHAR2 类型的值。如果指定了 n1,n2 那么从 n1 开始的长度为 n2 的字节将被返回。

EMPTY_BLOB()

该函数没有参数，函数返回 一个空的 BLOB 位置指示符。函数用于初始化一个 BLOB 变量或 BLOB 列。

EMPTY_CLOB()

该函数没有参数，函数返回 一个空的 CLOB 位置指示符。函数用于初始化一个 CLOB 变量或 CLOB 列。

GREATEST()

exp_list 是一列表达式，返回其中最大的表达式，每个表达式都被隐含的转换第一个表达式的数据类型，如果第一个表达式是字符串数据类型中的任何一个，那么返回的结果是 varchar2 数据类型， 同时使用的比较是非填充空格类型的比较。

LEAST()

exp_list 是一列表达式，返回其中最小的表达式，每个表达式都被隐含的转换第一个表达式的数据类型，如果第一个表达式是字符串数据类型中的任何一个，将返回的结果是 varchar2 数据类型， 同时使用的比较是非填充空格类型的比较。

UID

该函数没有参数，返回唯一标示当前数据库用户的整数。

USER

返回当前用户的用户名

USERENV()

基于 opt 返回包含当前会话信息。opt 的可选值为：

ISDBA 会话中 SYSDBA 脚色响应，返回 TRUE

SESSIONID 返回审计会话标示符

ENTRYID 返回可用的审计项标示符

INSTANCE 在会话连接后，返回实例标示符。该值只用于运行 Parallel 服务器并且有 多个实例的情况下使用。

LANGUAGE 返回语言、地域、数据库设置的字符集。

LANG 返回语言名称的 ISO 缩写。

TERMINAL 为当前会话使用的终端或计算机返回操作系统的标示符。

VSIZE()

x 是一个表达式。返回 x 内部表示的字节数。

单行函数和组函数

单行函数和组函数详解(5)

SQL 中的组函数

组函数也叫集合函数，返回基于多个行的单一结果，行的准确数量无法确定，除非查询被执行并且所有的结果都被包含在内。与单行函数不同的是，在解析时所有的行都是已知的。由于这种差别使组函数与单行函数有在要求和行为上有微小的差异.

组（多行）函数

与单行函数相比，oracle 提供了丰富的基于组的，多行的函数。这些函数可以在 select 或 select 的 having 子句中使用，当用于 select 子串时常常都和 GROUP BY 一起使用。

AVG([{DISYINCT|ALL}])

返回数值的平均值。缺省设置为 ALL.

SELECT AVG(sal),AVG(ALL sal),AVG(DISTINCT sal) FROM scott.emp

AVG(SAL) AVG(ALL SAL) AVG(DISTINCT SAL)

1877.94118 1877.94118 1916.071413

COUNT({*|DISTINCT|ALL})

返回查询中行的数目，缺省设置是 ALL,*表示返回所有的行。

MAX([{DISTINCT|ALL}])

返回选择列表项目的最大值，如果 x 是字符串数据类型，他返回一个 VARCHAR2 数据类型，如果 X 是一个 DATA 数据类型，返回一个日期，如果 X 是 numeric 数据类型，返回一个数字。注意 distinct 和 all 不起作用，应为最大值与这两种设置是相同的。

MIN([{DISTINCT|ALL}])

返回选择列表项目的最小值。

STDDEV([{DISTINCT|ALL}])

返回选者的列表项目的标准差，所谓标准差是方差的平方根。

S**UM([{DISTINCT|ALL}])**

返回选择列表项目的数值的总和。

VARIANCE([{DISTINCT|ALL}])

返回选择列表项目的统计方差。

用 GROUP BY 给数据分组

正如题目暗示的那样组函数就是操作那些已经分好组的数据，我们告诉数据库用 GROUP BY 怎样给数据分组或者分类，当我们在 SELECT 语句的 SELECT 子句中使用组函数时，我们必须把为分组或非常数列放置在 GROUP BY 子句中，如果没有用 group by 进行专门处理，那么缺省的分类是将整个结果设为一类。

```
select stat,counter(*) zip_count from zip_codes GROUP BY state;

ST ZIP_COUNT
-- ---------
AK 360
AL 1212
AR 1309
AZ 768
CA 3982
```

在这个例子中，我们用 state 字段分类；如果我们要将结果按照 zip_codes 排序,可以用 ORDER BY 语句，ORDER BY 子句可以使用列或组函数。

```
select stat,counter(*) zip_count from zip_codes GROUP BY state ORDER BY COUNT(*) DESC;

ST COUNT(*)
-- --------
NY 4312
PA 4297
TX 4123
CA 3982
```

用 HAVING 子句限制分组数据

现在你已经知道了在查询的 SELECT 语句和 ORDER BY 子句中使用主函数，组函数只能用于两个子串中，组函数不能用于 WHERE 子串中，例如下面的查询是错误的 ：

错误

```
SELECT sales_clerk,SUN(sale_amount) FROM gross_sales WHERE sales_dept='OUTSIDE' AND
SUM(sale_amount)>10000 GROUP BY sales_clerk
```

这个语句中数据库不知道 SUM()是什么，当我们需要指示数据库对行分组，然后限制分组后的行的输出时，正确的方法是使用 HAVING 语句：

```
SELECT sales_clerk,SUN(sale_amount)
```

```
FROM gross_sales
WHERE sales_dept='OUTSIDE'
GROUP BY sales_clerk
HAVING SUM(sale_amount)>10000;
```

嵌套函数

函数可以嵌套。一个函数的输出可以是另一个函数的输入。操作数有一个可继承的执行过程。但函数的优先权只是基于位置，函数遵循由内到外，由左到右的原则。嵌套技术一般用于象 DECODE 这样的能被用于逻辑判断语句 IF....THEN...ELSE 的函数。

嵌套函数可以包括在组函数中嵌套单行函数，或者组函数嵌套入单行函数或组函数中。比如下面的例子：

```
SELECT deptno, GREATEST(COUNT(DISTINCT job),COUNT(DISTINCT mgr) cnt,
COUNT(DISTINCT job) jobs,
COUNT(DISTINCT mgr) mgrs
FROM emp
GROUP BY deptno;

DEPTNO CNT JOBS MGRS
------ --- ---- ----
10 4 4 2
20 4 3 4
30 3 3 2
```

表和视图

表和视图(1)

Oracle 数据库数据对象中最基本的是表和视图，其他还有约束、序列、函数、存储过程、包、触发器等。对数据库的操作可以基本归结为对数据对象的操作,理解和掌握 Oracle 数据库对象是学习 Oracle 的捷径。

表和视图

Oracle 中表是数据存储的基本结构。ORACLE8 引入了分区表和对象表，ORACLE8i 引入了临时表，使表的功能更强大。视图是一个或多个表中数据的逻辑表达式。本文我们将讨论怎样创建和管理简单的表和视图。

管理表

表可以看作有行和列的电子数据表，表是关系数据库中一种拥有数据的结构。用 CREATE TABLE 语句建立表，在建立表的同时，必须定义表名，列，以及列的数据类型和大小。例如：

```
CREATE TABLE products
    ( PROD_ID NUMBER(4),
      PROD_NAME VAECHAR2(20),
      STOCK_QTY NUMBER(5,3)
    );
```

这样我们就建立了一个名为 products 的表， 关键词 CREATE TABLE 后紧跟的表名，然后定义了三列，同时规定了列的数据类型和大小。

在创建表的同时你可以规定表的完整性约束，也可以规定列的完整性约束，在列上普通的约束是 NOT NULL,关于约束的讨论我们在以后进行。

在建立或更改表时，可以给表一个缺省值。缺省值是在增加行时，增加的数据行中某一项值为 null 时，oracle 即认为该值为缺省值。

下列数据字典视图提供表和表的列的信息：

. DBA_TABLES

. DBA_ALL_TABLES

. USER_TABLES

. USER_ALL_TABLES

. ALL_TABLES

. ALL_ALL_TABLES

. DBA_TAB_COLUMNS

. USER_TAB_COLUMNS

. ALL_TAB_COLUMNS

表的命名规则

表名标识一个表，所以应尽可能在表名中描述表，oracle 中表名或列名最长可以达 30 个字符串。表名应该以字母开始，可以在表名中包含数字、下划线、#、$等。

从其它表中建立表

可以使用查询从基于一个或多个表中建立表，表的列的数据类型和大小有查询结果决

定。建立这种形式的表的查询可以选择其他表中所有的列或者只选择部分列。在 CREATE TABLE 语句中使用关键字 AS，例如:

```
SQL>CREATE TABLE emp AS SELECT * FROM employee

TABLE CREATED

SQL> CREATE TABLE Y AS SELECT * FROM X WHERE no=2
```

需要注意的是如果查询涉及 LONG 数据类型，那么 CREATE TABLE....AS SELECT.... 将不会工作。

更改表定义

在建立表后，有时候我们可能需要修改表，比如更改列的定义，更改缺省值，增加新列，删除列等等。ORACLE 使用 ALTER TABLE 语句来更改表的定义

1、增加列

语法：

```
ALTER TABLE [schema.] table_name ADD column_definition
```

例：

```
ALTER TABLE orders ADD order_date DATE;

TABLE ALTER
```

对于已经存在的数据行，新列的值将是 NULL.

2、更改列

语法：

```
ALTER TABLE [schema.] table_name MODIFY column_name new_attributes;
```

例：

```
ALTER TABLE orders MODITY (quantity number(10,3),status varchar2(15));
```

这个例子中我们修改了表 orders，将 STATUS 列的长度增加到 15,将 QUANTITY 列减小到 10,3;

修改列的规则如下：

. 可以增加字符串数据类型的列的长度，数字数据类型列的精度。

. 减少列的长度时，该列应该不包含任何值，所有数据行都为 NULL.

. 改变数据类型时，该列的值必须是 NULL.

. 对于十进制数字，可以增加或减少但不能降低他的精度。

3、删除数据列

优化 ORACLE 数据库，唯一的方法是删除列，重新建立数据库。在 ORACLE8i 中有很多方法删除列，你可以删除未用数据列或者可以标示该列为未用数据列然后删除。

删除数据列的语法是：

ALTER TABLE [schema.] table_name DROP {COLUM column_names | (column_names)}[CASCADE CONSTRAINS]

要注意的是在删除列时关于该列的索引和完整性约束也同时删除。注意关键字 CASCADE CONSTRAINS，如果删除的列是多列约束的一部分，那么这个约束条件相对于其他列也同时删除。

如果用户担心在大型数据库中删除列要花太多时间，可以先将他们标记为未用数据列，标记未用数据列的语法如下：

ALTER TABLE [schema.] table_name SET UNUSED {COLUM column_names | (column_names)}[CASCADE CONSTRAINS]

这个语句将一个或多个数据列标记为未用数据列，但并不删除数据列中的数据，也不释放占用的磁盘空间。但是，未用数据列在视图和数据字典中并不显示，并且该数据列的名称将被删除，新的数据列可以使用这个名称。基于该数据列的索引、约束，统计等都将被删除。

删除未用数据列的语句是：

ALTER TABLE [schema.] table_name DROP {UNUSED COLUM | COLUMN CONTINUE}

表和视图

表和视图(2)

删除表和更改表名

删除表非常简单，但它是一个不可逆转的行为。

语法：

```
DROP TABLE [schema.] table_name [CASCADE CONSTRAINTS]
```

删除表后，表上的索引、触发器、权限、完整性约束也同时删除。ORACLE 不能删除视图，或其他程序单元，但 oracle 将标示他们无效。如果删除的表涉及引用主键或唯一关键字的完整性约束时，那么 DROP TABLE 语句就必须包含 CASCADE CONSTRAINTS 子串。

更改表名

RENAME 命令用于给表和其他数据库对象改名。ORACLE 系统自动将基于旧表的完整性约束、索引、权限转移到新表中。ORACLE 同时使所有基于旧表的数据库对象，比如视图、程序、函数等，为不合法。

语法：

```
RENAME old_name TO new_name;
```

例：

```
SQL> RENAME orders TO purchase_orders;

TABLE RENAMED
```

截短表

TRUNCATE 命令与 DROP 命令相似， 但他不是删除整个数据表，所以索引、完整性

约束、触发器、权限等都不会被删除。缺省情况下将释放部分表和视图空间，如果用户不希望释放表空间，TRUNCATE 语句中要包含 REUSE STORAGE 子串。TRUNCATE 命令语法如下：

```
TRUNCATE {TABLE|CLUSTER} [schema.] name {DROP|REUSE STORAGE}
```

例：

```
SQL> TRUNCATE TABLE t1;

TABLE truncate.
```

管理视图

视图是一个或多个表中的数据的简化描述，用户可以将视图看成一个存储查询（stored query)或一个虚拟表(virtual table).查询仅仅存储在 oracle 数据字典中，实际的数据没有存放在任何其它地方，所以建立视图不用消耗其他的空间。视图也可以隐藏复杂查询，比如多表查询，但用户只能看见视图。视图可以有与他所基于表的列名不同的列名。用户可以建立限制其他用户访问的视图。

建立视图

CREATE VIEW 命令创建视图，定义视图的查询可以建立在一个或多个表，或其他视图上。查询不能有 FOR UPDATE 子串，在早期的 ORACLE8i 版本中不支持 ORDER BY 子串，现在的版本中 CREATE VIEW 可以拥有 ORDER BY 子串。

例：

```
SQL> CREATE VIEW TOP_EMP AS
SELECT empno EMPLOYEE_ID,ename EMPLOYEE_NAME,salary
FROM emp
WHERE salary >2000
```

用户可以在创建视图的同时更改列名，方法是在视图名后立即加上要命名的列名。重新定义视图需要包含 OR REPLACE 子串。

```
SQL> CREATE VIEW TOP_EMP
（EMPLOYEE_ID，EMPLOYEE_NAME，SALARY) AS
```

```
SELECT empno ,ename ,salary
FROM emp
WHERE salary >2000
```

如果在创建的视图包含错误在正常情况下，视图将不会被创建。但如果你需要创建一个带错误的视图必须在 CREATE VIEW 语句中带上 FORCE 选项。如：

```
CREATE FORCE VIEW ORDER_STATUS AS
SELECT * FROM PURCHASE_ORDERS
WHERE STATUS='APPPOVE';

SQL>/

warning :View create with compilation errors
```

这样将创建了一个名为 ORDER_STATUS 的视图，但这样的视图的状态是不合法的，如果以后状态发生变化则可以重新编译，其状态也变成合法的。

从视图中获得数据

从视图中获得数据与从表中获得数据基本一样，用户可以在连接和子查询中使用视图，也可以使用 SQL 函数，以及所有 SELECT 语句的字串。

插入、更新、删除数据

用户在一定的限制条件下可以通过视图更新、插入、删除数据。如果视图连接多个表，那么在一个时间里只能更新一个表。所有的能被更新的列可以在数据字典 USER_UPDATETABLE_COLUMNS 中查到。

用户在 CREATE VIEW 中可以使用了 WITH 子串。WITH READ ONLY 子串表示创建的视图是一个只读视图，不能进行更新、插入、删除操作。WITH CHECK OPTION 表示可以进行插入和更新操作，但应该满足 WHERE 子串的条件。这个条件就是创建视图 WHERE 子句的条件，比如在上面的例子中用户创建了一个视图 TOP_EMP,在这个视图中用户不能插入 salary 小于 2000 的数据行。

删除视图

删除视图使用 DROP VIEW 命令。同时将视图定义从数据字典中删除，基于视图的权限也同时被删除，其他涉及到该视图的函数、视图、程序等都将被视为非法。

例：

```
DROP VIEW TOP_EMP;
```

完整性约束

完整性约束(1)

完整性约束

完整性约束用于增强数据的完整性，Oracle 提供了 5 种完整性约束：

Check

NOT NULL

Unique

Primary

Foreign key

完整性约束是一种规则，不占用任何数据库空间。完整性约束存在数据字典中，在执行 SQL 或 PL/SQL 期间使用。用户可以指明约束是启用的还是禁用的，当约束启用时，他增强了数据的完整性，否则，则反之，但约束始终存在于数据字典中。

禁用约束，使用 ALTER 语句

```
ALTER TABLE table_name DISABLE CONSTRAINT constraint_name;
```

或

```
ALTER TABLE policies DISABLE CONSTRAINT chk_gender
```

如果要重新启用约束：

```
ALTER TABLE policies ENABLE CONSTRAINT chk_gender
```

删除约束

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name
```

或

```
ALTER TABLE policies DROP CONSTRAINT chk_gender;
```

Check 约束

在数据列上 Check 约束需要 一个特殊的布尔条件或者将数据列设置成 TRUE，至少一个数据列的值是 NULL，Check 约束用于增强表中数据内容的简单的商业规则。用户使用 Check 约束保证数据规则的一致性。Check 约束可以涉及该行同属 Check 约束的其他数据列但不能涉及其他行或其他表，或调用函数 SYSDATE,UID,USER,USERENV。如果用户的商业规则需要这类的数据检查，那么可以使用触发器。Check 约束不保护 LOB 数据类型的数据列和对象、嵌套表、VARRY、ref 等。单一数据列可以有多个 Check 约束保护，一

个 Check 约束可以保护多个数据列。

创建表的 Check 约束使用 CREATE TABLE 语句，更改表的约束使用 ALTER TABLE 语句。

语法：

```
CONSTRAINT [constraint_name] CHECK (condition);
```

Check 约束可以被创建或增加为一个表约束，当 Check 约束保护多个数据列时，必须使用表约束语法。约束名是可选的并且如果这个名字不存在，那么 oracle 将产生一个以 SYS_开始的唯一的名字。

例：

```
CREATE TABLE policies
(policy_id NUMBER,
holder_name VARCHAR2(40),
gender VARCHAR2(1) constraint chk_gender CHECK (gender in ('M','F'),
marital_status VARCHAR2(1),
date_of_birth DATE,
constraint chk_marital CHECK (marital_status in('S','M','D','W'))
);
```

NOT NULL 约束

NOT NULL 约束应用在单一的数据列上，并且他保护的数据列必须要有数据值。缺省状况下，ORACLE 允许任何列都可以有 NULL 值。某些商业规则要求某数据列必须要有值，NOT NULL 约束将确保该列的所有数据行都有值。

例：

```
CREATE TABLE policies
(policy_id NUMBER,
holder_name VARCHAR2(40) NOT NULL,
gender VARCHAR2(1),
marital_status VARCHAR2(1),
date_of_birth DATE NOT NULL
);
```

对于 NOT NULL 的 ALTER TABLE 语句与其他约束稍微有点不同。

```
ALTER TABLE policies MODIFY holder_name NOT NULL
```

完整性约束

完整性约束(2)

唯一性约束(Unique constraint)

唯一性约束可以保护表中多个数据列，保证在保护的数据列中任何两行的数据都不相同。唯一性约束与表一起创建，在唯一性约束创建后，可以使用 ALTER TABLE 语句修改。

语法：

```
column_name data_type CONSTRAINT constraint_name UNIQUE
```

如果唯一性约束保护多个数据列，那么唯一性约束要作为表约束增加。语法如下：

```
CONSTRAINT constraint_name (column) UNIQUE USING INDEX TABLESPACE (tablespace_name)
STORAGE (stored clause)
```

唯一性约束由一个 B-tree 索引增强，所以可以在 USING 子串中为索引使用特殊特征，比如表空间或存储参数。CREATE TABLE 语句在创建唯一性约束的同时也给目标数据列建立了一个唯一的索引。

```
CREATE TABLE insured_autos
(policy_id NUMBER CONSTRAINT pk_policies PRIMARY KEY,
vin VARCHAR2(10),
coverage_begin DATE,
coverage_term NUMBER,
CONSTRAIN unique_auto UNIQUE (policy_id,vin) USING INDEX TABLESPACE index STORAGE
(INITIAL 1M NEXT 10M PCTINCREASE 0)
);
```

用户可以禁用未以性约束，但他仍然存在，禁用唯一性约束使用 ALTER TABLE 语句

```
ALTER TABLE insured_autos DISABLE CONSTRAIN unique_name;
```

删除唯一性约束，使用 ALTER TABLE....DROP CONSTRAIN 语句

```
ALTER TABLE insured_autos DROP CONSTRAIN unique_name;
```

注意用户不能删除在有外部键指向的表的唯一性约束。这种情况下用户必须首先禁用或删除外部键（foreign key)。

删除或禁用唯一性约束通常同时删除相关联的唯一索引，因而降低了数据库性能。经常删除或禁用唯一性约束有可能导致丢失索引带来的性能错误。要避免这样错误，可以采取下面的步骤：

1、在唯一性约束保护的数据列上创建非唯一性索引。

2、添加唯一性约束

主键(Primary Key)约束

表有唯一的主键约束。表的主键可以保护一个或多个列，主键约束可与 NOT NULL 约束共同作用于每一数据列。NOT NULL 约束和唯一性约束的组合将保证主键唯一地标识每一行。像唯一性约束一样，主键由 B-tree 索引增强。

创建主键约束使用 CREATE TABLE 语句与表一起创建，如果表已经创建了，可以使用 ALTER TABLE 语句。

```
CREATE TABLE policies
(policy_id NUMBER CONSTRAINT pk_policies PRIMARY KEY,
holder_name VARCHAR2(40),
gender VARCHAR2(1),
marital_status VARCHAR2(1),
date_of_birth DATE
);
```

与唯一性约束一样，如果主键约束保护多个数据列，那么必须作为一个表约束创建。

```
CREATE TABLE insured_autos
(policy_id NUMBER,
vin VARCHAR2(40),
coverage_begin DATE,
coverage_term NUMBER,
CONSTRAINT pk_insured_autos PRIMARY KEY (policy_id,vin)
USING INDEX TABLESPACE index
STORAGE (INITIAL 1M NEXT 10M PCTINCREASE 0)
);
```

禁用或删除主键必须与 ALTER TABLE 语句一起使用

```
ALTER TABLE policies DROP PRIMARY KEY;
```

或

```
ALTER TABLE policies DISABLE PRIMARY KEY;
```

外部键约束（Foreign key constraint)

外部键约束保护一个或多个数据列，保证每个数据行的数据包含一个或多个 null 值，或者在保护的数据列上同时拥有主键约束或唯一性约束。引用（主键或唯一性约束）约束可以保护同一个表，也可以保护不同的表。与主键和唯一性约束不同外部键不会隐式建立一个 B-tree 索引。在处理外部键时，我们常常使用术语父表（parent table)和子表（child table), 父表表示被引用主键或唯一性约束的表，子表表示引用主键和唯一性约束的表。

创建外部键使用 CREATE TABLE 语句，如果表已经建立了，那么使用 ALTER TABLE 语句。

```
CREATE TABLE insured_autos
(policy_id NUMBER CONSTRAINT policy_fk
REFERENCE policies(policy_id
ON DELETE CASCADE,
vin VARCHAR2(40),
coverage_begin DATE,
coverage_term NUMBER,
make VARCHAR2(30),
model VARCHAR(30),
year NUMBER,
CONSTRAIN auto_fk FROEIGN KEY (make,model,year)
REFERENCES automobiles (make,model,year)
```

```
ON DELETE SET NULL
);
```

ON DELETE 子串告诉 ORACLE 如果父纪录（parent record)被删除后，子记录做什么。缺省情况下禁止在子记录还存在的情况下删除父纪录。

外部键和 NULL 值

在外部键约束保护的数据列中 NULL 值的处理可能产生不可预料的结果。ORACLE 使用 ISO standar Match None 规则增强外部键约束。这个规则规定如果任何外部键作用的数据列包含有一个 NULL 值，那么任何保留该键的数据列在父表中没有匹配值。

比如，在父表 AUTOMOBILES 中，主键作用于数据列 MAKE，MODEL，YEAR 上，用户使用的表 INSURED_AUTOS 有一个外部约束指向 AOTOMOBILES，注意在 INSURES_AUTOS 中有一数据行的 MODEL 列为 NULL 值，这一行数据已经通过约束检查，即使 MAKE 列也没有显示在父表 AUTOMOBILES 中，如下表：

表 1 AUTOMOBILES

MAKE	MODEL	YEAR
Ford	Taurus	2000
Toyota	Camry	1999

表 2 INSURED_AUTOS

POLICY_ID	MAKE	MODEL	YEAR
576	Ford	Taurus	2000
577	Toyota	Camry	1999

578	Tucker	NULL	1949

延迟约束检验（Deferred Constraint Checking）

约束检验分两种情况，一种是在每一条语句结束后检验数据是否满足约束条件，这种检验称为立即约束检验（immediately checking）,另一种是在事务处理完成之后对数据进行检验称之为延迟约束检验。在缺省情况下 Oracle 约束检验是立即检验（immediately checking), 如果不满足约束将先是一条错误信息，但用户可以通过 SET CONSTRAINT 语句选择延迟约束检验。语法如下：

SET CONSTRAINT constraint_name|ALL DEFEERRED|IMMEDIATE --;

完整性约束

完整性约束(3)

序列（Sequences）

Oracle 序列是一个连续的数字生成器。序列常用于人为的关键字，或给数据行排序否则数据行是无序的。像约束一样，序列只存在于数据字典中。序列号可以被设置为上升、下降，可以没有限制或重复使用直到一个限制值。创建序列使用 SET SEQUENCE 语句。

CREATE SEQUENCE [schema] sequence KEYWORD

KEYWORD 包括下面的值：

KEYWORD	描述
START WITH	定义序列生成的第一个数字，缺省为 1

INCREMENT BY	定义序列号是上升还是下降，对于一个降序的序列 INCREMENT BY 为负值
MINVALUE	定义序列可以生成的最小值，这是降序序列中的限制值。缺省情况下该值为 NOMINVALUE,NOMINVALUE，对于升序为 1，对于降序为-10E26.
MAXVALUE	序列能生成的最大数字。这是升序序列中的限制值，缺省的 MAXVALUE 为 NOMAXVALUE,NOMAXVALUE，对于升序为 10E26，对于降序为-1。
CYCLE	设置序列值在达到限制值以后可以重复
NOCYCLE	设置序列值在达到限制值以后不能重复，这是缺省设置。当试图产生 MAXVALUE+1 的值时，将会产生一个异常
CACHE	定义序列值占据的内存块的大小，缺省值为 20
NOCACHE	在每次序列号产生时强制数据字典更新，保证在序列值之间没有间隔当创建序列时，START WITH 值必须等于或大于 MINVALUE。

删除序列使用 DROP SEQUENCE 语句

```
DROP SEQUENCE sequence_name
```

索引(INDEXES)

索引是一种可以提高查询性能的数据结构，在这一部分我们将讨论索引如何提高查询性能的。ORACLE 提供了以下几种索引：

B-Tree、哈希（hash)、位图（bitmap)等索引类型
基于原始表的索引
基于函数的索引
域（Domain）索引

实际应用中主要是 B-Tree 索引和位图索引，所以我们将集中讨论这两种索引类型。

B-Tree 索引

B-Tree 索引是最普通的索引，缺省条件下建立的索引就是这种类型的索引。B-Tree 索引可以是唯一或非唯一的，可以是单一的（基于一列）或连接的（多列）。B-Tree 索引在检索高基数数据列（高基数数据列是指该列有很多不同的值）时提供了最好的性能。对于取出较小的数据 B-Tree 索引比全表检索提供了更有效的方法。但当检查的范围超过表的 10%时就不能提高取回数据的性能。正如名字所暗示的那样，B-Tree 索引是基于二元树的，由枝干块（branch block)和树叶块（leaf block)组成,枝干块包含了索引列（关键字）和另一索

引的地址。树叶块包含了关键字和给表中每个匹配行的 ROWID。

位图索引

位图索引主要用于决策支持系统或静态数据，不支持行级锁定。位图索引可以是简单的（单列）也可以是连接的（多列），但在实践中绝大多数是简单的。位图索引最好用于低到中群集（cardinality)列，在这些列上多位图索引可以与 AND 或 OR 操作符结合使用。位图索引使用位图作为键值，对于表中的每一数据行位图包含了 TRUE（1）、FALSE（0）、或 NULL 值。位图索引的位图存放在 B-Tree 结构的页节点中。B-Tree 结构使查找位图非常方便和快速。另外，位图以一种压缩格式存放，因此占用的磁盘空间比 B-Tree 索引要小得多。

同义词（Synonyms）

对另一个数据对象而言同义词是一个别名。public 同义词是针对所有用户的，相对而言 private 同义词则只针对对象拥有者或被授予权限的账户。在本地数据库中同义词可以表示表、视图、序列、程序、函数或包等数据对象，也可以通过链接表示另一个数据库的对象。

创建同义词语法如下：

```
CREATE [PUBLIC] SYNONYM synonym_name FOR [schema.] object[@db_link];
```

例：

```
CREATE PUBLIC SYNONYM policies FOR poladm.policies@prod;

CREATE SYNONYM plan_table FOR system.plan_table;
```

过程和函数

过程和函数(1)

过程和函数

过程和函数都以编译后的形式存放在数据库中，函数可以没有参数也可以有多个参数并有一个返回值。过程有零个或多个参数，没有返回值。函数和过程都可以通过参数列表接收或返回零个或多个值，函数和过程的主要区别不在于返回值，而在于他们的调用方式。过程是作为一个独立执行语句调用的：

```
pay_involume(invoice_nbr,30,due_date);
```

函数以合法的表达式的方式调用：

```
order_volumn:=open_orders(SYSDATE,30);
```

创建过程的语法如下：

```
CREATE [ OR REPLACE] PROCEDURE [schema.]procedure_name
[parameter_lister]
{AS|IS}
declaration_section
BEGIN
executable_section
[EXCEPTION
exception_section]
END [procedure_name]
```

每个参数的语法如下：

```
paramter_name mode datatype [(:=|DEFAULT) value]
```

mode 有三种形式：IN、OUT、INOUT。

IN 表示在调用过程的时候，实际参数的取值被传递给该过程，形式参数被认为是只读的，当过程结束时，控制会返回控制环境，实际参数的值不会改变。

OUT 在调用过程时实际参数的取值都将被忽略，在过程内部形式参数只能是被赋值，而不能从中读取数据，在过程结束后形式参数的内容将被赋予实际参数。

INOUT 这种模式是 IN 和 OUT 的组合；在过程内部实际参数的值会传递给形式参数，形势参数的值可读也可写，过程结束后，形势参数的值将赋予实际参数。

创建函数的语法和过程的语法基本相同，唯一的区别在于函数有 RETUREN 子句

```
CREATE [ OR REPLACE] FINCTION [schema.]function_name
[parameter_list]
RETURN returning_datatype
{AS|IS}
declaration_section
BEGIN
executable_section
[EXCEPTION]
exception_section
END [procedure_name]
```

在执行部分函数必须有哟个或多个 return 语句。

在创建函数中可以调用单行函数和组函数，例如：

```
CREATE OR REPLACE FUNCTION my_sin(DegreesIn IN NUMBER)
RETURN NUMBER
IS
```

```
pi NUMBER=ACOS(-1);
RadiansPerDegree NUMBER;

BEGIN
RadiansPerDegree=pi/180;
RETURN(SIN(DegreesIn*RadiansPerDegree));
END
```

过程和函数

过程和函数(2)

包

包是一种将过程、函数和数据结构捆绑在一起的容器；包由两个部分组成：外部可视包规范，包括函数头，过程头，和外部可视数据结构；另一部分是包主体(package body),包主体包含了所有被捆绑的过程和函数的声明、执行、异常处理部分。

打包的 PL/SQL 程序和没有打包的有很大的差异，包数据在用户的整个会话期间都一直存在，当用户获得包的执行授权时，就等于获得包规范中的所有程序和数据结构的权限。但不能只对包中的某一个函数或过程进行授权。包可以重载过程和函数，在包内可以用同一个名字声明多个程序，在运行时根据参数的数目和数据类型调用正确的程序。

创建包必须首先创建包规范，创建包规范的语法如下：

```
CREATE [OR REPLACE] PACKAGE package_name
{AS|IS}
public_variable_declarations |
public_type_declarations |
public_exception_declarations |
public_cursor_declarations |
function_declarations |
procedure_specifications
END [package_name]
```

创建包主体使用 CREATE PACKAGE BODY 语句：

```
CREATE [OR REPLACE] PACKAGE BODY package_name
{AS|IS}
private_variable_declarations |
private_type_declarations |
private_exception_declarations |
private_cursor_declarations |
function_declarations |
procedure_specifications
END [package_name]
```

私有数据结构是那些在包主体内部，对被调用程序而言是不可见的。

触发器(Triggers)

触发器是一种自动执行响应数据库变化的程序。可以设置为在触发器事件之前或之后触发或执行。能够触发触发器事件的事件包括下面几种：

DML 事件
DDL 事件
数据库事件

DML 事件触发器可以是语句或行级触发器。DML 语句触发器在触发语句之前或之后触发 DML 行级触发器在语句影响的行变化之前或之后触发。用户可以给单一事件和类型定义多个触发器，但没有任何方法可以增强多触发器触发的命令。下表列出了用户可以利用的触发器事件：

事件	触发器描述
INSERT	当向表或视图插入一行时触发触发器
UPDATE	更新表或视图中的某一行时触发触发器
DELETE	从表或视图中删除某一行时触发触发器
CREATE	当使用 CREATE 语句为数据库或项目增加一个对象时触发触发器
ALTER	当使用 ALTER 语句为更改一个数据库或项目的对象时触发触发器
DROP	当使用 DROP 语句删除一个数据库或项目的对象时触发触发器
START	打开数据库时触发触发器，在事件后触发
SHUTDOWN	关闭数据库时触发，事件前触发

LOGON	当一个会话建立时触发，事件前触发
LOGOFF	当关闭会话时触发，事件前触发
SERVER	服务器错误发生时触发触发器，事件后触发

创建触发器的语法如下：

```
CREATE [OR REPLACE] TRIGGER trigger_name
{before|after|instead of} event
ON {table_or_view_name|DATABASE}
[FOR EACH ROW[WHEN condition]]
trigger_body
```

只有 DML 触发器（INSERT、UPDATE、DELETE)语句可以使用 INSTEAD OF 触发器并且只有表的 DML 触发器可以是 BEFORE 或 AFTER 触发器。

象约束一样触发器可以被设置为禁用或启用来关闭或打开他们的执行体(EXECUTE), 将触发器设置为禁用或启用使用 ALTER TRIGGER 语句：

```
ALTER TRIGGER trigger_name ENABLE;
ALTER TRIGGER trigger_name DISABLE;
```

要禁用或启用表的所有触发器，使用 ALTER TABLE 语句

```
ALTER TRIGGER table_name DISABLE ALL TRIGGER;
ALTER TRIGGER table_name ENABLE ALL TRIGGER;
```

删除触发器使用 DROP TRIGGER

```
DROP TRIGGER trigger_name;
```

数据字典

Oracle 数据字典包含了用户数据库的元数据。带下划线的表名称中带 OBJ$、UET$、SOURCE$，这些表是在执行 CREATE DATABASE 语句期间由 sql.bsq 脚本创建的，一般情况下用户很少访问这些表。脚本 catalog.sql（通常位于$oracle_home/rdbms/admin）在 CREATE DATABASE 语句之后立即运行，创建数据字典视图。

数据字典视图大致可以分为三类：

.前缀为 USER_的数据字典视图，包含了用户拥有的对象的信息。

.前缀为 ALL_的数据字典视图，包含了用户当前可以访问的全部对象和权限的信息。

.前缀为 DBA_的数据字典视图，包含了数据库拥有的所有对象和权限的信息。

在绝大多数数据字典视图中都有象 DBA_TABLES,ALL_TABLES 和 USER_TABLES 这样的视图家族。Oracle 中有超过 100 个视图家族，所以要全面介绍这些视图家族是单调乏味的而且没有多大的意义。在下表中列出了最重要和最常用的视图家族，需要注意的是每个视图家族都有一个 DBA_,一个 ALL_一个 USER_视图。

视图家族(View Family)	描述
COL_PRIVS	包含了表的列权限，包括授予者、被授予者和权限
EXTENTS	数据范围信息，比如数据文件，数据段名(segment_name)和大小
INDEXES	索引信息，比如类型、唯一性和被涉及的表
IND_COLUMNS	索引列信息，比如索引上的列的排序方式
OBJECTS	对象信息，比如状态和 DDL time
ROLE_PRIVS	角色权限，比如 GRANT 和 ADMIN 选项
SEGMENTS	表和索引的数据段信息，比如 tablespace 和 storage
SEQUECNCES	序列信息，比如序列的 cache、cycle 和 ast_number
SOURCE	除触发器之外的所有内置过程、函数、包的源代码
SYNONYMS	别名信息，比如引用的对象和数据库链接 db_link
SYS_PRIVS	系统权限，比如 grantee、privilege、admin 选项
TAB_COLUMNS	表和视图的列信息，包括列的数据类型
TAB_PRIVS	表权限，比如授予者、被授予者和权限
TABLES	表信息，比如表空间(tablespace),存储参数(storage parms)和数据行的数量
TRIGGERS	触发器信息，比如类型、事件、触发体(trigger body)
USERS	用户信息，比如临时的和缺省的表空间
VIEWS	视图信息，包括视图定义

在 Oracle 中还有一些不常用的数据字典表，但这些表不是真正的字典家族，他们都是一些重要的单一的视图。

VIEW NAME	描述
USER_COL_PRIVS_MADE	用户授予他人的列权限
USER_COL_PRIVS_RECD	用户获得的列权限

USER_TAB_PRIVS_MADE	用户授予他人的表权限
USER_TAB_PRIVS_RECD	用户获得的表权限

其他的字典视图中主要的是 V$视图，之所以这样叫是因为他们都是以 V$或 GV$开头的。V$视图是基于 X$虚拟视图的。V$视图是 SYS 用户所拥有的，在缺省状况下，只有 SYS 用户和拥有 DBA 系统权限的用户可以看到所有的视图，没有 DBA 权限的用户可以看到 USER_和 ALL_视图，但不能看到 DBA_视图。与 DBA_,ALL,和 USER_视图中面向数据库信息相反，这些视图可视的给出了面向实例的信息。

在大型系统上化几周时间手工输入每一条语句

手工输入带用户名变量的语句，然后再输入每一个用户名，这需要花好几个小时的时间

写一条 SQL 语句，生成需要的 ALTER USER 语句，然后执行他，这只需要几分钟时间

很明显我们将选择生成 SQL 的方法：

例：

```
SELECT 'ALTER USER'||username||
'TEMPORARY TABLESPACE temp;'
FROM DBA_USERS
WHERE username<>'SYS'
AND temporary_tablespace<>'TEMP';
```

这个查询的结果将被脱机处理到一个文件中，然后在执行：

```
ALTER USER SYSTEM TEMPORARY TABLESPACE temp;
ALTER USER OUTLN TEMPORARY TABLESPACE temp;
ALTER USER DBSNMP TEMPORARY TABLESPACE temp;
ALTER USER SCOTT TEMPORARY TABLESPACE temp;
ALTER USER DEMO TEMPORARY TABLESPACE temp;
```

操作和控制语言

数据操作和控制语言详解(1)

SQL 语言共分为四大类:数据查询语言 DQL,数据操纵语言 DML，数据定义语言 DDL，数据控制语言 DCL。其中用于定义数据的结构，比如 创建、修改或者删除数据库；DCL 用于定义数据库用户的权限；在这篇文章中我将详细讲述这两种语言在 Oracle 中的使用方法。

DML 语言

DML 是 SQL 的一个子集，主要用于修改数据，下表列出了 ORACLE 支持的 DML 语句。

语句	用途
INSERT	向表中添加行
UPDATE	更新存储在表中的数据
DELETE	删除行
SELECT FOR UPDATE	禁止其他用户访问 DML 语句正在处理的行。
LOCK TABLE	禁止其他用户在表中使用 DML 语句

插入数据

INSERT 语句常常用于向表中插入行，行中可以有特殊数据字段，或者可以用子查询从已存在的数据中建立新行。

列目录是可选的，缺省的列的目录是所有的列名，包括 comlumn_id,comlumn_id 可以在数据字典视图 ALL_TAB_COLUMNS,USER_TAB_COLUMNS,或者

DBA_TAB_COLUMNS 中找到。

插入行的数据的数量和数据类型必须和列的数量和数据类型相匹配。不符合列定义的数据类型将对插入值实行隐式数据转换。NULL 字符串将一个 NULL 值插入适当的列中。关键字 NULL 常常用于表示将某列定义为 NULL 值。

下面的两个例子是等价的。

```
INSERT INTO customers(cust_id,state,post_code)
VALUE('Ariel',NULL,'94501');
```

或

```
INSERT INTO customers(cust_id,state,post_code)
VALUE('Ariel',,'94501');
```

更新数据

UPDATE 命令用于修改表中的数据。

```
UPDATE order_rollup
SET(qty,price)=(SELECT SUM(qty),SUM(price) FROM order_lines WHERE customer_id='KOHL'
WHERE cust_id='KOHL'
AND order_period=TO_DATE('01-Oct-2000')
```

删除数据

DELETE 语句用来从表中删除一行或多行数据，该命令包含两个语句：

1、关键字 DELETE FROM 后跟准备从中删除数据的表名。

2、WHERE 后跟删除条件

```
DELETE FROM po_lines
WHERE ship_to_state IN ('TX','NY','IL')
AND order_date<TRUNC(SYSTEM)-90< td>
```

清空表

如果你想删除表中所有数据，清空表，可以考虑使用 DDL 语言的 TRUNCATE 语句。TRUNCATE 就像没有 WHERE 子句的 DELETE 命令一样。TRUNCATE 将删除表中所有行。TRUNCATE 不是 DML 语句是 DDL 语句，他和 DELETE 右不同的特点。

```
TRUNCATE TABLE (schema)table DROP(REUSE) STORAGE
```

STORAGE 子串是可选的，缺省是 DROP STORAGE。当使用 DROP STORAGE 时将缩短表和表索引，将表收缩到最小范围，并重新设置 NEXT 参数。REUSE STORAGE 不会缩短表或者调整 NEXT 参数。

TRUNCATE 和 DELETE 有以下几点区别

1、TRUNCATE 在各种表上无论是大的还是小的都非常快。如果有 ROLLBACK 命令 DELETE 将被撤销，而 TRUNCATE 则不会被撤销。

2、TRUNCATE 是一个 DDL 语言，向其他所有的 DDL 语言一样，他将被隐式提交，不能对 TRUNCATE 使用 ROLLBACK 命令。

3、TRUNCATE 将重新设置高水平线和所有的索引。在对整个表和索引进行完全浏览时，经过 TRUNCATE 操作后的表比 DELETE 操作后的表要快得多。

4、TRUNCATE 不能触发任何 DELETE 触发器。

5、不能授予任何人清空他人的表的权限。

6、当表被清空后表和表的索引讲重新设置成初始大小，而 delete 则不能。

7、不能清空父表。

SELECT FOR UPDATE

select for update 语句用于锁定行，阻止其他用户在该行上修改数据。当该行被锁定后其他用户可以用 SELECT 语句查询该行的数据，但不能修改或锁定该行。

锁定表

LOCK 语句常常用于锁定整个表。当表被锁定后，大多数 DML 语言不能在该表上使用。LOCK 语法如下：

```
LOCK schema table IN lock_mode
```

其中 lock_mode 有两个选项：

share 共享方式

exclusive 唯一方式

例：

LOCK TABLE intentory IN EXCLUSIVE MODE

死锁

当两个事务都被锁定，并且互相都在等待另一个被解锁，这种情况称为死锁。

当出现死锁时，ORACLE 将检测死锁条件，并返回一个异常。

操作和控制语言

数据操作和控制语言详解(2)

事务控制

事务控制包括协调对相同数据的多个同步的访问。当一个用户改变了另一个用户正在使用的数据时，oracle 使用事务控制谁可以操作数据。

事务

事务表示工作的一个基本单元，是一系列作为一个单元被成功或不成功操作的 SQL 语句。在 SQL 和 PL/SQL 中有很多语句让程序员控制事务。程序员可以：

1、显式开始一个事物，选择语句级一致性或事务级一致性

2、设置撤销回滚点，并回滚到回滚点

3、完成事务永远改变数据或者放弃修改。

事务控制语句

语句	用途
Commit	完成事务，数据修改成功并对其他用户开放
Rollback	撤销事务，撤销所有操作
rollback to savepoint	撤销在设置的回滚点以后的操作
set transaction	响应事务或语句的一致性；特别对于事务使用回滚段

例：

```
BEGIN
UPDATE checking
SET balance=balance-5000
WHERE account='Kieesha';

INSERT INTO checking_log(action_date,action,amount)
VALUES (SYSDATE,'Transfer to brokerage',-5000);

UPDATE brokerage
SET cash_balance=cash_balance+5000
WHERE account='Kiesha';

INSERT INTO brokerage_log(action_date,action,amount)
VALUES (SYSDATE,'Tracfer from checking',5000)

COMMIT

EXCEPTION
WHEN OTHERS
ROLLBACK

END
```

Savepoint 和 部分回滚(Partial Rollback)

在 SQL 和 PL/SQL 中 Savepoint 是在一事务范围内的中间标志。经常用于将一个长的

事务划分为小的部分。保留点 Savepoint 可标志长事务中的任何点，允许可回滚该点之后的操作。在应用程序中经常使用 Savepoint；例如一过程包含几个函数，在每个函数前可建立一个保留点，如果函数失败，很容易返回到每一个函数开始的情况。在回滚到一个 Savepoint 之后，该 Savepoint 之后所获得的数据封锁被释放。为了实现部分回滚可以用带 TO Savepoint 子句的 ROLLBACK 语句将事务回滚到指定的位置。

例

```
BEGIN

INSERT INTO ATM_LOG(who,when,what,where)
VALUES ('Kiesha',SYSDATE,'Withdrawal of $100','ATM54')
SAVEPOINT ATM_LOGGED;

UPDATE checking
SET balance=balance-100
RETURN balance INTO new_balance;

IF new_balance<0
THEN
ROLLBACK TO ATM_LOGGED;
COMMIT
RAISE insufficient_funda;
END IF

END
```

关键字 SAVEPOINT 是可选的，所以下面两个语句是等价的：

```
ROLLBACK TO ATM_LOGGED;
ROLLBACK TO SAVEPOINT ATM_LOGGED;
```

一致性和事务

一致性是事物控制的关键概念。掌握了 oracle 的一致性模型，能使您更好的，更恰当的使用事务控制。oracle 通过一致性保证数据只有在事务全部完成后才能被用户看见和使用。这项技术对多用户数据库有巨大的作用。

oracle 常常使用语句级（state-level)一致性，保证数据在语句的生命期之间是可见的但不能被改变。事务由多个语句组成，当使用事务时，事物级（transaction-level)一致性在整

个事务生命期中保证数据对所有语句都是可见的。

oracle 通过 SCN(syatem change number)实施一致性。一个 SCN 是一个面向时间的数据库内部键。SCN 只会增加不会减少，SCN 表示了时间上的一个点，每个数据块都有一个 SCN,通过比较这个点实施操作。

事务级一致性

SET TRANSACTION 的一个作用是确保事务级一致或语句级一致中有一个实施。ORACLE 使用这些术语：

ISOLATION LEVEL READ COMMIT 表示语句级一致

ISOLATION LEVEL SERIALIZABLE 表示事务级一致。

例：

```
SET TRANSACTION ISOLATION LEVEL READ COMMIT;

SET TRANSACTION ISOLATION LEVEL READ COMMIT
```

下面的语句也能确保事务级一致：

```
SET TRANSCATION READ ONLY
```

任何企图在只读（READ ONLY）事务中修改数据的操作都会抛出一个异常。但是，READ ONLY 事务只能在下列语句中使用：

```
SELECT(没有 FOR UPDATE 子句）
LOCK TABLE
SET ROLE
ALTER SYSTEM
ALTER ALARM
```

即使没有改变任何数据，READ ONLY 事务依然必须使用一个 COMMIT 或 ROLLBACK 以结束整个事务。

SET TRANSCTION 的另外一个应用是在回滚时直接使用回滚段（ROLLBACK

SEGMENT）。回滚段是 ORACLE 的一个特殊的数据对象，回滚段的头部包含正在使用该回滚段事务的信息。当用户回滚事务（ROLLBACK）时，ORACLE 将会利用回滚段中的数据前影像来将修改的数据恢复到原来的值。oracle 用 round-robin 给事务随机分配回滚段。一个大的事务可以分配任何回滚段，这也许会导致回滚段的大小变得很大。因此要避免让大的事务随机分配回滚段。

事务以 SET TRANSACTION 开始,象下面这样：

SET TRANSACTION USE ROLLBACK SEGMENT rb_large;

rb_large 是一个大的回滚段的名称，现在就给一个大的事务分配了一个大的回滚段，其他的小的回滚段将不由动态空间管理，这样就更有效率。

下面我们看一个例子.我们有一个回滚段表空间大小是 2G，在高峰时期需要 10 个回滚段以满足用户的需要，这些高峰在线用户只有小的事务。一周我们连续运行了 4 个大的事务，这些事务需要删除和加载数据，每一个撤销需要 1G，回滚段的大小如下：

rb_large(initial 100M minextenta 2)

rb1 (initial 1M next minextents 5)
rb2 (initial 1M next minextents 5)
rb3 (initial 1M next minextents 5)
rb4 (initial 1M next minextents 5)
rb5 (initial 1M next minextents 5)
rb6 (initial 1M next minextents 5)
rb7 (initial 1M next minextents 5)
rb8 (initial 1M next minextents 5)
rb9 (initial 1M next minextents 5)
rb10 (initial 1M next minextents 5)

所有的都非常恰当的安排在 2G 的表空间中，如果我们缺省的 round-robin 给事务分配回滚段，4 个大事务将有 4 个独立的回滚段，每个回滚段的大小将是 1G,如果这样我们的 2G 表空间就不够，而数据库管理员就不得不在夜晚 2 点起来工作，每个事务都由以下面的语句开始：

SET TRANSACTION USE ROLLBACK SEGMENT rb_large

现在 4 个事务重用相同的表空间，保正 4 个回滚段的表空间在 2G 以内。数据库管理员可以睡到天亮。

操作和控制语言

数据操作和控制语言详解(3)

建立和修改用户

CREATE USER 语句将建立一个用户。当一个用户连接到 ORACLE 数据库时，它必须被验证。ORACLE 中验证有三种类型：

Database

external

Global

缺省是数据库验证，当用户连接到数据库时，oracle 将检测用户是否是数据库的合法用户，并且要提供正确的 password.external 验证，oracle 将只检测用户是否是合法用户，password 已经被网络或系统验证了。global 验证也是只检测是否是合法用户，password 由 oraclesecurity server 验证。

Database 验证用户账号

数据库验证账号是张好的缺省类型，也是最普通的类型。建立一个账号是 piyush,口令是 welcome 的账号，只需执行下面的命令：

CREATE USE piyush IDENTIFIED BY welcome

piyush 可以通过下面的语句将口令改变为 saraswatt:

```
ALTER USER piyush IDENTIFIED BY saraswati;
```

外部验证用户账号

用户账号进入数据库时可以不提供口令，这种情况下代替数据库识别口令的是客户端操作系统。外部验证账号有时也叫 OPS$账号，当他们最初在 oracle6 开始介绍时，oracle 账号都有关键字前缀 OPS$,这也就是为什么 init.ora 参数 os_authent_prefix 是 OPS$--默认特征与 oracle6 保持一致。os_authent_prefix 定义的字符串必须被预处理为用于 Oracle 外部识别账号的操作系统账号名。创建操作系统用户 appl 的语句是：

```
CREATE USER ops$appl IDENTIFIED EATERNALLY
```

但在通常情况下，os_authent_prefix 将被设置为空，像下面这样：

```
CREATE USER appl IDENTIFIED EATERNALLY
```

这样效果是一样的,关键字 IDENTIFIED EXTERNALLY 告诉 ORACLE 这是一个外部识别账号。

GLOBAL 用户账号

GLOBAL 类型的用户账号数据库不检测口令，而是由 X.509 目录服务器检测口令。创建一个 GLOBAL 类型的用户账号的方法是：

```
CREATE USER scott IDENTIFIED GLOBALLY AS "CN=scott,OU=divisional,O=sybex,C=US"
```

关键字 IDENTIFIED GLOBALLY AS 表示建立的是一个 GLOBAL 类型的用户账号.

创建和更改用户账号

CREATE USER 用于建立用户账号和给用户账号的属性赋值。ALTER USER 用于更改用户账号和属性。但 CREATE USER 语句必须包括用户名和口令。

有部分属性能用 CREATER USER 和 ALTER USER 语句设置，下面对是这些的属性

具体描述：

给用户分配缺省表空间

表空间（tablespace)是放置表、索引、丛等用户对象的。如果在 create user 语句中没有包含表空间，那么缺省的是系统表空间。

```
CREATE USER piyush IDENTIFIED BY saraswati
DEFAULTE TABLESPACE user_data;
ALTER USER manoj DEFAULTE TABLESPACE dev1_data;
```

给用户分配临时表空间

临时表空间，顾名思义是临时存放表、索引等用户对象的临时段。建立方法一样

```
CREATE USER piyush IDENTIFIED BY saraswati
Temporary TABLESPACE user_data;
ALTER USER manoj Temporary TABLESPACE dev1_data;
```

给用户分配表空间的使用定额

使用定额限制用户在表空间中使用磁盘的数量。定额可以按字节、千字节、兆字节或者无限制来制定。

```
CREATE USER piyush IDENTIFIED BY saraswati
DEFAULT TABLESPACE user_data
QUOTA UNLIMITED ON user_data
QUOTA 20M ON tools;
ALTER USER manoj QUOTA 2500K ON tools;
```

给用户分配一个简表

简表可以限制用户在会话时消耗的资源。这些资源包括：连接数据库的时间，空闲时间，每次会话的逻辑读数据的数量等等，缺省的简表对资源无限制。

```
CREATE USER piyush IDENTIFIED BY saraswati
PROFILE TABLESPACE user_data;
ALTER USER manoj Temporary TABLESPACE dev1_data;
```

为用户响应指定角色

这个属性只能由 ALTER USER 语句设置，试图用 CREATE USER 语句设置将回返回一个例外。

```
ALTER USER manoj DEFAULT ROLE ALL EXCEPT salary_adm;
```

为用户的 password 设定到期时间以便在用户下次登录时更改

当用户的 password 到期，在下一次登录时将强迫修改 password，oracle 提示用户输入旧的 password,然后输入新的 password。这项功能常用于新用户，当新用户用缺省的 password 登录时必须修改立即修改 password.

```
ALTER USER manoj IDENTIFIED BY welcome;
ALTER USER manoj PASSWORD EXPIRE;
```

锁定账号，是用户不能登录

```
ALTER USER ql AC
COUNT LOCK
```

对账号解锁，以便用户能登录数据库

```
ALTER USER ql ACCOUNT UNLOCK
```

权限和角色

权限允许用户访问属于其它用户的对象或执行程序，ORACLE 系统提供三种权限：

Object 对象级

System 系统级

Role 角色级

这些权限可以授予给用户、特殊用户 public 或角色，如果授予一个权限给特殊用户

"Public"（用户 public 是 oracle 预定义的，每个用户享有这个用户享有的权限）,那么就意味作将该权限授予了该数据库的所有用户。

对管理权限而言，角色是一个工具，权限能够被授予给一个角色，角色也能被授予给另一个角色或用户。用户可以通过角色继承权限，除了管理权限外角色服务没有其它目的。权限可以被授予，也可以用同样的方式撤销。

建立和使用角色

如前所诉，角色存在的目的就是为了使权限的管理变得轻松。建立角色使用 CREATE ROLE 语句，他的语法如下：

```
CREATE ROLE role_name IDENTIFIED BY password
CREATE ROLE role_name IDENTIFIED EXTERNALLY
CREATE ROLE role_name IDENTIFIED GLOBALLY
```

缺省情况下建立的角色没有 password 或者其他的识别。如果使用 IDENTIFIED BY 子句建立，那么角色不会自动响应，必须用 SET ROLE 激活。

```
SET ROLE role_name IDENTIFIED BY password
```

EXTERNALLY 和 GLOBALLY 类型的角色由操作系统和 ORACLE Service server 验证。通常用户需要权限修改应用程序中使用的表单中的数据，但是只有在应用程序运行时而不是在使用 ad hoc 工具时，这种上下文敏感安全可以通过有 PASSWORD 的角色来实现。当用户在应用程序内部连结数据库时，代码将执行 SET ROLE 命令，通过安全验证。所以用户不需要知道角色的 password,也不需要自己输入 SET ROLE 命令。

对象权限

对象权限就是指在表、视图、序列、过程、函数或包等对象上执行特殊动作的权利。有九种不同类型的权限可以授予给用户或角色。如下表：

权限	ALTER	DELETE	EXECUTE	INDEX	INSERT	READ	REFERENCE	SELECT	UPDATE
Directory	no	no	no	no	no	yes	no	no	no
function	no	no	yes	no	no	no	no	no	no
procedure	no	no	yes	no	no	no	no	no	no
package	no	no	yes	no	no	no	no	no	no
DB Object	no	no	yes	no	no	no	no	no	no

Libary	no	no	yes	no	no	no	no	no	no
Operation	no	no	yes	no	no	no	no	no	no
Sequence	yes	no	no	no	no	no	no	no	no
Table	yes	yes	no	yes	yes	no	yes	yes	yes
Type	no	no	yes	no	no	no	no	no	no
View	no	yes	no	no	yes	no	no	yes	yes

对象由不止一个权限，特殊权限 ALL 可以被授予或撤销。如 TABLE 的 ALL 权限就包括：

SELECT,INSERT,UPDATE 和 DELETE,还有 INDEX,ALTER,和 REFERENCE。

如何看这个表我们以 ALTER 权限为例进行说明

ALTER 权限

允许执行 ALTER TABLE 和 LOCK TABLE 操作，ALTER TABLE 可以进行如下操作：

. 更改表名

. 增加或删除列

. 改变列的数据类型或大小

. 将表转变为分区表

在 SEQUENCE 上的 ALTER 权限允许执行 ALTER Sequence 语句，重新给 sequence 分配最小值、增量和缓冲区大小。

系统权限

系统权限需要授予者有进行系统级活动的能力，如连接数据库，更改用户会话、建立表或建立用户等等。你可以在数据字典视图 SYSTEM_PRIVILEGE_MAP 上获得完整的系统权限。对象权限和系统权限都通过 GRANT 语句授予用户或角色。需要注意的是在授予对象权限时语句应该是 WITH GRANT OPTION 子句，但在授予系统权象时语句是 WITH ADMIN OPTION，所以在你试图授予系统权限时，使用语句 WITH GRANT OPTION 系统会报告一

个错误：ONLY ADMIN OPTION can be specified。在考试中要特别注意这个语法和错误信息。

角色和角色权限

角色权限就是将属于用户的权限授予一个角色。任何权限都可以授予给一个角色。授予系统权限给被授予者必须使用 WITH_ADMIN_OPTION 子句，在会话期间通过 SET ROLE 语句授予或撤销角色权限。然而，角色权限不能依靠存储在 SQL 中的权限。如果函数、程序、包、触发器或者方法使用另一个计划拥有的对象，那么就必须直接给对象的拥有者授权，这是因为权限不会在会话之间改变。

授予和撤销权限

给用户或者角色授予权限使用 GRANT 语句,GRANT 语句的语法如下：

GRANT ROLE（或 system privilege） TO user(role,Public) WITH ADMIN OPTION（可选）

对象权限被授予 WITH GRANT OPTION，

权限和数据字典

数据字典是 ORACLE 存储有关数据库结构信息的地方，数据本身存放在其他地方，数据字典由表和视图组成。在考试中关于数据字典最容易考的内容是：查看那一类权限已经被授予。比如 DBA_TAB_PRIV 包含了用户授予给另一用户的对象权限和在授予时是否带有 WITH GRANT OTPION 子串的信息。注意 DBA_TAB_PRIV 不仅仅包含了对表的权限的关系，他还包括函数、包、队列等等上的权限的关系。下表列出了所有的权限和角色的数据字典视图：

表： 权限的数据字典视图

视图	作用
ALL_COL_PRIVS	表示列上的授权，用户和 PUBLIC 是被授予者
ALL_COL_PRIVS_MADE	表示列上的授权，用户是属主和被授予者
ALL_COL_RECD	表示列上的授权，用户和 PUBLIC 是被授予者
ALL_TAB_PRIVS	表示对象上的授权，用户是 PUBLIC 或被授予者或用户是属主
ALL_TAB_PRIVS_MADE	表示对象上的权限，用户是属主或授予者
ALL_TAB_PRIVS_RECD	表示对象上的权限, 用户是 PUBLIC 或被授予者

DBA_COL_PRIVS	数据库列上的所有授权
DBA_ROLE_PRIVS	显示已授予用户或其他角色的角色
DBA_SYS_PRIVS	已授予用户或角色的系统权限
DBA_TAB_PRIVS	数据库对象上的所有权限
ROLE_ROLE_PRIVS	显示已授予用户的角色
ROLE_SYS_PRIVS	显示通过角色授予用户的系统权限
ROLE_TAB_PRIVS	显示通过角色授予用户的对象权限
SESSION_PRIVS	显示用户现在可利用的所有系统权限
USER_COL_PRIVS	显示列上的权限，用户是属主、授予者或被授予者
USER_COL_PRIVS_MADE	显示列上已授予的权限，用户是属主或授予者
USER_COL_PRIVS_RECD	显示列上已授予的权限，用户是属主或被授予者
USER_ROLE_PRIVS	显示已授予给用户的所有角色
USER_SYS_PRIVS	显示已授予给用户的所有系统权限
USER_TAB_PRIVS	显示已授予给用户的所有对象权限
USER_TAB_PRIVS_MADE	显示已授予给其他用户的对象权限，用户是属主
USER_TAB_PRIVS_RECD	显示已授予给其他用户的对象权限，用户是被授予者

游标

游标使用大全(1)

SQL 是用于访问 ORACLE 数据库的语言，PL/SQL 扩展和加强了 SQL 的功能，它同时引入了更强的程序逻辑。PL/SQL 支持 DML 命令和 SQL 的事务控制语句。DDL 在 PL/SQL 中不被支持，这就意味作在 PL/SQL 程序块中不能创建表或其他任何对象。较好的 PL/SQL 程序设计是在 PL/SQL 块中使用象 DBMS_SQL 这样的内建包或执行 EXECUTE IMMEDIATE 命令建立动态 SQL 来执行 DDL 命令，PL/SQL 编译器保证对象引用以及用户的权限。

下面我们将讨论各种用于访问 ORACLE 数据库的 DDL 和 TCL 语句。

查询

SELECT 语句用于从数据库中查询数据，当在 PL/SQL 中使用 SELECT 语句时，要与 INTO 子句一起使用，查询的返回值被赋予 INTO 子句中的变量，变量的声明是在 DELCARE 中。SELECT INTO 语法如下：

```
SELECT [DISTICT|ALL]{*|column[,column,...]}
INTO (variable[,variable,...] |record)
FROM {table|(sub-query)}[alias]
WHERE.....
```

PL/SQL 中 SELECT 语句只返回一行数据。如果超过一行数据，那么就要使用显式游标（对游标的讨论我们将在后面进行），INTO 子句中要有与 SELECT 子句中相同列数量的变量。INTO 子句中也可以是记录变量。

%TYPE 属性

在 PL/SQL 中可以将变量和常量声明为内建或用户定义的数据类型，以引用一个列名，同时继承他的数据类型和大小。这种动态赋值方法是非常有用的，比如变量引用的列的数据类型和大小改变了，如果使用了%TYPE,那么用户就不必修改代码，否则就必须修改代码。

例：

```
v_empno SCOTT.EMP.EMPNO%TYPE;
v_salary EMP.SALARY%TYPE;
```

不但列名可以使用%TYPE,而且变量、游标、记录，或声明的常量都可以使用%TYPE。这对于定义相同数据类型的变量非常有用。

```
DELCARE
V_A NUMBER(5):=10;
V_B V_A%TYPE:=15;
V_C V_A%TYPE;
BEGIN
DBMS_OUTPUT.PUT_LINE
('V_A='||V_A||'V_B='||V_B||'V_C='||V_C);
END

SQL>/
V_A=10 V_B=15 V_C=
PL/SQL procedure successfully completed.

SQL>
```

其他 DML 语句

其它操作数据的 DML 语句是:INSERT、UPDATE、DELETE 和 LOCK TABLE,这些语句在 PL/SQL 中的语法与在 SQL 中的语法相同。我们在前面已经讨论过 DML 语句的使用这里就不再重复了。在 DML 语句中可以使用任何在 DECLARE 部分声明的变量，如果是嵌套块，那么要注意变量的作用范围。

例：

```
CREATE OR REPLACE PROCEDURE FIRE_EMPLOYEE (pempno in number)
  AS
    v_ename EMP.ENAME%TYPE;
  BEGIN
    SELECT ename INTO v_ename
    FROM emp
    WHERE empno=p_empno;

    INSERT INTO FORMER_EMP(EMPNO,ENAME)
    VALUES (p_empno,v_ename);

    DELETE FROM emp
    WHERE empno=p_empno;
```

```
    UPDATE former_emp
    SET date_deleted=SYSDATE
    WHERE empno=p_empno;

    EXCEPTION
      WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Employee Number Not Found!');

END
```

DML 语句的结果

当执行一条 DML 语句后，DML 语句的结果保存在四个游标属性中，这些属性用于控制程序流程或者了解程序的状态。当运行 DML 语句时，PL/SQL 打开一个内建游标并处理结果，游标是维护查询结果的内存中的一个区域，游标在运行 DML 语句时打开，完成后关闭。隐式游标只使用 SQL%FOUND,SQL%NOTFOUND,SQL%ROWCOUNT 三个属性.SQL%FOUND,SQL%NOTFOUND 是布尔值，SQL%ROWCOUNT 是整数值。

SQL%FOUND 和 SQL%NOTFOUND

在执行任何 DML 语句前 SQL%FOUND 和 SQL%NOTFOUND 的值都是 NULL,在执行 DML 语句后，SQL%FOUND 的属性值将是：

. TRUE :INSERT

. TRUE :DELETE 和 UPDATE，至少有一行被 DELETE 或 UPDATE.

. TRUE :SELECT INTO 至少返回一行

当 SQL%FOUND 为 TRUE 时,SQL%NOTFOUND 为 FALSE。

SQL%ROWCOUNT

在执行任何 DML 语句之前，SQL%ROWCOUNT 的值都是 NULL,对于 SELECT INTO 语句，如果执行成功，SQL%ROWCOUNT 的值为 1,如果没有成功，SQL%ROWCOUNT 的值为 0，同时产生一个异常 NO_DATA_FOUND.

SQL%ISOPEN

SQL%ISOPEN 是一个布尔值，如果游标打开，则为 TRUE, 如果游标关闭，则为 FALSE. 对于隐式游标而言 SQL%ISOPEN 总是 FALSE，这是因为隐式游标在 DML 语句执行时打开，结束时就立即关闭。

事务控制语句

事务是一个工作的逻辑单元可以包括一个或多个 DML 语句，事物控制帮助用户保证数据的一致性。如果事务控制逻辑单元中的任何一个 DML 语句失败，那么整个事务都将回滚，在 PL/SQL 中用户可以明确地使用 COMMIT、ROLLBACK、SAVEPOINT 以及 SET TRANSACTION 语句。

COMMIT 语句终止事务，永久保存数据库的变化，同时释放所有 LOCK,ROLLBACK 终止现行事务释放所有 LOCK，但不保存数据库的任何变化,SAVEPOINT 用于设置中间点，当事务调用过多的数据库操作时，中间点是非常有用的，SET TRANSACTION 用于设置事

务属性，比如 read-write 和隔离级等。

显式游标

当查询返回结果超过一行时，就需要一个显式游标，此时用户不能使用 select into 语句。PL/SQL 管理隐式游标，当查询开始时隐式游标打开，查询结束时隐式游标自动关闭。显式游标在 PL/SQL 块的声明部分声明，在执行部分或异常处理部分打开，取数据,关闭。下表显示了显式游标和隐式游标的差别：

表 1 隐式游标和显式游标

隐式游标	显式游标
PL/SQL 维护，当执行查询时自动打开和关闭	在程序中显式定义、打开、关闭，游标有一个名字。
游标属性前缀是 SQL	游标属性的前缀是游标名
属性%ISOPEN 总是为 FALSE	%ISOPEN 根据游标的状态确定值
SELECT 语句带有 INTO 子串，只有一行数据被处理	可以处理多行数据，在程序中设置循环，取出每一行数据。

游标

游标使用大全(2)

使用游标

这里要做一个声明，我们所说的游标通常是指显式游标，因此从现在起没有特别指明的情况，我们所说的游标都是指显式游标。要在程序中使用游标，必须首先声明游标。

声明游标

语法：

```
CURSOR cursor_name IS select_statement;
```

在 PL/SQL 中游标名是一个未声明变量，不能给游标名赋值或用于表达式中。

例：

```
DELCARE
CURSOR C_EMP IS SELECT empno,ename,salary
FROM emp
WHERE salary>2000
ORDER BY ename;
.....
BEGIN
```

在游标定义中 SELECT 语句中不一定非要表可以是视图，也可以从多个表或视图中选择的列，甚至可以使用*来选择所有的列 。

打开游标

使用游标中的值之前应该首先打开游标，打开游标初始化查询处理。打开游标的语法是：

```
OPEN cursor_name
```

cursor_name 是在声明部分定义的游标名。

例：

```
OPEN C_EMP;
```

关闭游标

语法：

```
CLOSE cursor_name
```

例：

```
CLOSE C_EMP;
```

从游标提取数据

从游标得到一行数据使用 FETCH 命令。每一次提取数据后，游标都指向结果集的下一行。语法如下：

```
FETCH cursor_name INTO variable[,variable,...]
```

对于 SELECT 定义的游标的每一列，FETCH 变量列表都应该有一个变量与之相对应，变量的类型也要相同。

例：

```
SET SERVERIUTPUT ON
DECLARE
v_ename EMP.ENAME%TYPE;
v_salary EMP.SALARY%TYPE;
CURSOR c_emp IS SELECT ename,salary FROM emp;
BEGIN
OPEN c_emp;
```

```
FETCH c_emp INTO v_ename,v_salary;
DBMS_OUTPUT.PUT_LINE('Salary of Employee'|| v_ename
||'is'|| v_salary);
FETCH c_emp INTO v_ename,v_salary;
DBMS_OUTPUT.PUT_LINE('Salary of Employee'|| v_ename
||'is'|| v_salary);
FETCH c_emp INTO v_ename,v_salary;
DBMS_OUTPUT.PUT_LINE('Salary of Employee'|| v_ename
||'is'|| v_salary);
CLOSE c_emp;
END
```

这段代码无疑是非常麻烦的，如果有多行返回结果，可以使用循环并用游标属性为结束循环的条件，以这种方式提取数据，程序的可读性和简洁性都大为提高，下面我们使用循环重新写上面的程序：

```
SET SERVERIUTPUT ON
DECLARE
v_ename EMP.ENAME%TYPE;
v_salary EMP.SALARY%TYPE;
CURSOR c_emp IS SELECT ename,salary FROM emp;
BEGIN
OPEN c_emp;
LOOP
FETCH c_emp INTO v_ename,v_salary;
EXIT WHEN c_emp%NOTFOUND;
DBMS_OUTPUT.PUT_LINE('Salary of Employee'|| v_ename
||'is'|| v_salary);
END
```

记录变量

定义一个记录变量使用 TYPE 命令和%ROWTYPE，关于%ROWsTYPE 的更多信息请参阅相关资料。

记录变量用于从游标中提取数据行，当游标选择很多列的时候，那么使用记录比为每列

声明一个变量要方便得多。

当在表上使用%ROWTYPE 并将从游标中取出的值放入记录中时，如果要选择表中所有列，那么在 SELECT 子句中使用*比将所有列名列出来要安全得多。

例：

```
SET SERVERIUTPUT ON
DECLARE
R_emp EMP%ROWTYPE;
CURSOR c_emp IS SELECT * FROM emp;
BEGIN
OPEN c_emp;
LOOP
FETCH c_emp INTO r_emp;
EXIT WHEN c_emp%NOTFOUND;
DBMS_OUT.PUT.PUT_LINE('Salary of Employee'||r_emp.ename||'is'|| r_emp.salary);
END LOOP;
CLOSE c_emp;
END;
```

%ROWTYPE 也可以用游标名来定义，这样的话就必须要首先声明游标：

```
SET SERVERIUTPUT ON
DECLARE
CURSOR c_emp IS SELECT ename,salary FROM emp;
R_emp c_emp%ROWTYPE;
BEGIN
OPEN c_emp;
LOOP
FETCH c_emp INTO r_emp;
EXIT WHEN c_emp%NOTFOUND;
DBMS_OUT.PUT.PUT_LINE('Salary of Employee'||r_emp.ename||'is'|| r_emp.salary);
END LOOP;
CLOSE c_emp;
END;
```

带参数的游标

与存储过程和函数相似，可以将参数传递给游标并在查询中使用。这对于处理在某种条件下打开游标的情况非常有用。它的语法如下：

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

定义参数的语法如下：

```
Parameter_name [IN] data_type[{:=|DEFAULT} value]
```

与存储过程不同的是，游标只能接受传递的值，而不能返回值。参数只定义数据类型，没有大小。

另外可以给参数设定一个缺省值，当没有参数值传递给游标时，就使用缺省值。游标中定义的参数只是一个占位符，在别处引用该参数不一定可靠。

在打开游标时给参数赋值，语法如下：

```
OPEN cursor_name[value[,value]....];
```

参数值可以是文字或变量。

例：

```
DECALRE
CURSOR c_dept IS SELECT * FROM dept ORDER BY deptno;
```

```
CURSOR c_emp (p_dept VARACHAR2) IS
SELECT ename,salary
FROM emp
WHERE deptno=p_dept
ORDER BY ename
r_dept DEPT%ROWTYPE;
v_ename EMP.ENAME%TYPE;
v_salary EMP.SALARY%TYPE;
v_tot_salary EMP.SALARY%TYPE;

BEGIN

OPEN c_dept;
LOOP
FETCH c_dept INTO r_dept;
EXIT WHEN c_dept%NOTFOUND;
DBMS_OUTPUT.PUT_LINE('Department:'|| r_dept.deptno||'-'||r_dept.dname);
v_tot_salary:=0;
OPEN c_emp(r_dept.deptno);
LOOP
FETCH c_emp INTO v_ename,v_salary;
EXIT WHEN c_emp%NOTFOUND;
DBMS_OUTPUT.PUT_LINE('Name:'|| v_ename||' salary:'||v_salary);
v_tot_salary:=v_tot_salary+v_salary;
END LOOP;
CLOSE c_emp;
DBMS_OUTPUT.PUT_LINE('Toltal Salary for dept:'|| v_tot_salary);
END LOOP;
CLOSE c_dept;
END;
```

游标

游标使用大全(3)

游标 FOR 循环

在大多数时候我们在设计程序的时候都遵循下面的步骤：

1、打开游标

2、开始循环

3、从游标中取值

4、检查那一行被返回

5、处理

6、关闭循环

7、关闭游标

可以简单的把这一类代码称为游标用于循环。但还有一种循环与这种类型不相同，这就是 FOR 循环，用于 FOR 循环的游标按照正常的声明方式声明，它的优点在于不需要显式的打开、关闭、取数据，测试数据的存在、定义存放数据的变量等等。游标 FOR 循环的语法如下：

```
FOR record_name IN
(corsor_name[(parameter[,parameter]...)]
| (query_difinition)
LOOP
statements
END LOOP;
```

下面我们用 for 循环重写上面的例子：

```
DECALRE

CURSOR c_dept IS SELECT deptno,dname FROM dept ORDER BY deptno;
CURSOR c_emp (p_dept VARACHAR2) IS
SELECT ename,salary
FROM emp
WHERE deptno=p_dept
ORDER BY ename
```

```
v_tot_salary EMP.SALARY%TYPE;

BEGIN

FOR r_dept IN c_dept LOOP
DBMS_OUTPUT.PUT_LINE('Department:'|| r_dept.deptno||'-'||r_dept.dname);
v_tot_salary:=0;
FOR r_emp IN c_emp(r_dept.deptno) LOOP
DBMS_OUTPUT.PUT_LINE('Name:'|| v_ename||' salary:'||v_salary);
v_tot_salary:=v_tot_salary+v_salary;
END LOOP;
DBMS_OUTPUT.PUT_LINE('Toltal Salary for dept:'|| v_tot_salary);
END LOOP;

END;
```

在游标 FOR 循环中使用查询

在游标 FOR 循环中可以定义查询，由于没有显式声明所以游标没有名字，记录名通过游标查询来定义。

```
DECALRE

  v_tot_salary EMP.SALARY%TYPE;

BEGIN

  FOR r_dept IN (SELECT deptno,dname FROM dept ORDER BY deptno) LOOP
    DBMS_OUTPUT.PUT_LINE('Department:'|| r_dept.deptno||'-'||r_dept.dname);
    v_tot_salary:=0;
    FOR r_emp IN (SELECT ename,salary
      FROM emp
      WHERE deptno=p_dept
      ORDER BY ename) LOOP
    DBMS_OUTPUT.PUT_LINE('Name:'|| v_ename||' salary:'||v_salary);
    v_tot_salary:=v_tot_salary+v_salary;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Toltal Salary for dept:'|| v_tot_salary);
  END LOOP;

END;
```

游标中的子查询

语法如下：

```
CURSOR C1 IS SELECT * FROM emp
WHERE deptno NOT IN (SELECT deptno
FROM dept
WHERE dname!='ACCOUNTING');
```

可以看出与 SQL 中的子查询没有什么区别。

游标中的更新和删除

在 PL/SQL 中依然可以使用 UPDATE 和 DELETE 语句更新或删除数据行。显式游标只有在需要获得多行数据的情况下使用。PL/SQL 提供了仅仅使用游标就可以执行删除或更新记录的方法。

UPDATE 或 DELETE 语句中的 WHERE CURRENT OF 子串专门处理要执行 UPDATE 或 DELETE 操作的表中取出的最近的数据。要使用这个方法，在声明游标时必须使用 FOR UPDATE 子串，当对话使用 FOR UPDATE 子串打开一个游标时，所有返回集中的数据行都将处于行级（ROW-LEVEL)独占式锁定，其他对象只能查询这些数据行，不能进行 UPDATE、DELETE 或 SELECT...FOR UPDATE 操作。

语法：

```
FOR UPDATE [OF [schema.]table.column[,[schema.]table.column]..
[nowait]
```

在多表查询中，使用 OF 子句来锁定特定的表,如果忽略了 OF 子句，那么所有表中选择的数据行都将被锁定。如果这些数据行已经被其他会话锁定，那么正常情况下 ORACLE 将等待，直到数据行解锁。

在 UPDATE 和 DELETE 中使用 WHERE CURRENT OF 子串的语法如下：

```
WHERE{CURRENT OF cursor_name|search_condition}
```

例：

```
DELCARE

CURSOR c1 IS SELECT empno,salary
FROM emp
WHERE comm IS NULL
FOR UPDATE OF comm;

v_comm NUMBER(10,2);

BEGIN

FOR r1 IN c1 LOOP

IF r1.salary<500 THEN
v_comm:=r1.salary*0.25;
ELSEIF r1.salary<1000 THEN
v_comm:=r1.salary*0.20;
ELSEIF r1.salary<3000 THEN
v_comm:=r1.salary*0.15;
ELSE
v_comm:=r1.salary*0.12;
END IF;

UPDATE emp;
SET comm=v_comm
WHERE CURRENT OF c1l;

END LOOP;
END
```

异常处理

异常处理初步(1)

PL/SQL 处理异常不同于其他程序语言的错误管理方法，PL/SQL 的异常处理机制与 ADA 很相似，有一个处理错误的全包含方法。当发生错误时，程序无条件转到异常处理部分，这就要求代码要非常干净并把错误处理部分和程序的其它部分分开。oracle 允许声明其他异常条件类型以扩展错误/异常处理。这种扩展使 PL/SQL 的异常处理非常灵活。

当一个运行时错误发生时，称为一个异常被抛出。PL/SQL 程序编译时的错误不是能被处理得异常，只有在运行时的异常能被处理。在 PL/SQL 程序设计中异常的抛出和处理是非常重要的内容。

抛出异常

由三种方式抛出异常

. 通过 PL/SQL 运行时引擎

. 使用 RAISE 语句

. 调用 RAISE_APPLICATION_ERROR 存储过程

当数据库或 PL/SQL 在运行时发生错误时，一个异常被 PL/SQL 运行时引擎自动抛出。异常也可以通过 RAISE 语句抛出

RAISE exception_name;

显式抛出异常是程序员处理声明的异常的习惯用法，但 RAISE 不限于声明了的异常，

它可以抛出任何任何异常。例如，你希望用 TIMEOUT_ON_RESOURCE 错误检测新的运行时异常处理器，你只需简单的在程序中使用下面的语句：

RAISE TIMEOUT_ON_RESOUCE;

下面看一个订单输入系统，当库存小于订单时抛出一个 inventory_too_low 异常。

```
DECLARE
inventory_too_low EXCEPTION;
---其他声明语句
BEGIN
.
.
IF order_rec.qty>inventory_rec.qty THEN
RAISE inventory_too_low;
END IF
.
.
EXCEPTION
WHEN inventory_too_low THEN
order_rec.staus:='backordered';
replenish_inventory(inventory_nbr=>
inventory_rec.sku,min_amount=>order_rec.qty-inventory_rec.qty);
END;
```

这里 replenish_inventory 是一个触发器。

处理异常

PL/SQL 程序块的异常部分包含了程序处理错误的代码，当异常被抛出时，一个异常陷阱就自动发生，程序控制离开执行部分转入异常部分,一旦程序进入异常部分就不能再回到同一块的执行部分。下面是异常部分的一般语法：

```
EXCEPTION
  WHEN exception_name THEN
    Code for handing exception_name
  [WHEN another_exception THEN
    Code for handing another_exception]
  [WHEN others THEN
    code for handing any other exception.]
```

用户必须在独立的 WHEN 子串中为每个异常设计异常处理代码，WHEN OTHERS 子串必须放置在最后面作为缺省处理器处理没有显式处理的异常。当异常发生时，控制转到异常部分，ORACLE 查找当前异常相应的 WHEN..THEN 语句，捕捉异常，THEN 之后的代码被执行，如果错误陷阱代码只是退出相应的嵌套块，那么程序将继续执行内部块 END 后面的语句。如果没有找到相应的异常陷阱，那么将执行 WHEN OTHERS。在异常部分 WHEN 子串没有数量限制。

```
EXCEPTION

  WHEN inventory_too_low THEN
    order_rec.staus:='backordered';
    replenish_inventory(inventory_nbr=>
    inventory_rec.sku,min_amount=>order_rec.qty-inventory_rec.qty);
  WHEN discontinued_item THEN
    --code for discontinued_item processing
  WHEN zero_divide THEN
    --code for zero_divide
  WHEN OTHERS THEN
    --code for any other exception
END;
```

当异常抛出后，控制无条件转到异常部分，这就意味着控制不能回到异常发生的位置，当异常被处理和解决后，控制返回到上一层执行部分的下一条语句。

```
BEGIN
  DECLARE
    bad_credit;
  BEGIN
```

```
    RAISE bad_credit;
      --发生异常，控制转向；
  EXCEPTION
    WHEN bad_credit THEN
      dbms_output.put_line('bad_credit');
    END;

    --bad_credit 异常处理后，控制转到这里
  EXCEPTION
    WHEN OTHERS THEN
      --控制不会从 bad_credit 异常转到这里
      --因为 bad_credit 已被处理
END;
```

当异常发生时，在块的内部没有该异常处理器时，控制将转到或传播到上一层块的异常处理部分。

```
BEGIN
  DECLARE ---内部块开始
    bad_credit;
  BEGIN
    RAISE bad_credit;
      --发生异常，控制转向；
    EXCEPTION
    WHEN ZERO_DIVIDE THEN --不能处理 bad_credite 异常
      dbms_output.put_line('divide by zero error');
    END --结束内部块

      --控制不能到达这里，因为异常没有解决；
      --异常部分

    EXCEPTION
    WHEN OTHERS THEN
      --由于 bad_credit 没有解决，控制将转到这里
END;
```

异常传播

没有处理的异常将沿检测异常调用程序传播到外面，当异常被处理并解决或到达程序最外层　ネＶ埂?BR>在声明部分抛出的异常将控制转到上一层的异常部分。

```
BEGIN
executable statements
BEGIN
today DATE:='SYADATE'; --ERRROR
BEGIN --内部块开始
dbms_output.put_line('this line will not execute');
EXCEPTION
WHEN OTHERS THEN
--异常不会在这里处理
END;--内部块结束

EXCEPTION
WHEN OTHERS THEN
处理异常
END
```

执行部分抛出的异常将首先传递到同一块的异常部分，如果在同一块的异常部分没有处理这个异常的处理器，那么异常将会传播到上一层的异常部分中，一直到最外层。

在异常部分抛出的异常将控制转到上一层的异常部分。

处理异常将停止异常的传播和解决。有时用户希望在错误发生时，程序仍然能执行一些动作，要达到这个目的，可以把希望执行的动作放在异常处理器中，然后执行不带参数的 RAISE 语句，RAISE 语句将重新抛出出现的异常，允许他继续传播。

```
DECLARE
order_too_old EXCEPTION;
BEGIN
RAISE order_too_old;
EXCEPTION
WHEN order_too_old THEN
DECLARE
```

```
file_handle UTL_FILE.FILE_TYPE;
BEGIN
--open file
file_handle:=UTL_FILE.FOPEN
(location=>'/ora01/app/oracle/admin/test/utlsir'
,filename=>'error.log'
.open_mode=>'W');
--write error stack
UTL_FILE.PUT_LINE(filehandle,
DBMS_UTILITY.FORMAT_ERROR_STACK);
--write the call stack
UTL_FILE.PUT_LINE(filehandle,
DBMS_UTILITY.FORMAT_CALL_STACK);
--close error log
UTL_FILE.FCLOSE(file_handle);
RAISE; --re-raise the exception
END
END
```

如果从 FORMAT_XXX_STACK 输出一个很大的值，那么使用 DBMS_OUTPUT 或 UTL_FILE 显示错误或调用堆的异常部分自身也会抛出异常，这两个堆常规下最多能返回 2000 字节，但 utl_file.put_line 被限制在 1000 字节以内，而 dbms_output.put_line 限制在 512 字节内。如果使用前面的代码并且不允许这种可能性，那么在异常处理器中将抛出一个未处理的异常。

GOTO 语句不能用于将控制从执行部分传递到异常部分或反之。

已命名异常

在 PL/SQL 块的异常部分只有已命名的异常才能被 WHEN 子串处理，ORACLE 包含了一系列已命名的异常，这些异常都声明在 STANDARD 包中，这些内建异常在这里就不一一讲述，有兴趣的读者可以查阅有关资料。

异常处理

异常处理初步(2)

PL/SQL 处理异常不同于其他程序语言的错误管理方法，PL/SQL 的异常处理机制与 ADA 很相似，有一个处理错误的全包含方法。当发生错误时，程序无条件转到异常处理部分，这就要求代码要非常干净并把错误处理部分和程序的其它部分分开。oracle 允许声明其他异常条件类型以扩展错误/异常处理。这种扩展使 PL/SQL 的异常处理非常灵活。

当一个运行时错误发生时，称为一个异常被抛出。PL/SQL 程序编译时的错误不是能被处理得异常，只有在运行时的异常能被处理。在 PL/SQL 程序设计中异常的抛出和处理是非常重要的内容。

抛出异常

由三种方式抛出异常

. 通过 PL/SQL 运行时引擎

. 使用 RAISE 语句

. 调用 RAISE_APPLICATION_ERROR 存储过程

当数据库或 PL/SQL 在运行时发生错误时，一个异常被 PL/SQL 运行时引擎自动抛出。异常也可以通过 RAISE 语句抛出

RAISE exception_name;

显式抛出异常是程序员处理声明的异常的习惯用法，但 RAISE 不限于声明了的异常，它可以抛出任何任何异常。例如，你希望用 TIMEOUT_ON_RESOURCE 错误检测新的运行时异常处理器，你只需简单的在程序中使用下面的语句：

RAISE TIMEOUT_ON_RESOUCE;

下面看一个订单输入系统，当库存小于订单时抛出一个 inventory_too_low 异常。

```
DECLARE
inventory_too_low EXCEPTION;
---其他声明语句
BEGIN
.
.
IF order_rec.qty>inventory_rec.qty THEN
RAISE inventory_too_low;
END IF
.
.
EXCEPTION
WHEN inventory_too_low THEN
order_rec.staus:='backordered';
replenish_inventory(inventory_nbr=>
inventory_rec.sku,min_amount=>order_rec.qty-inventory_rec.qty);
END;
```

这里 replenish_inventory 是一个触发器。

处理异常

PL/SQL 程序块的异常部分包含了程序处理错误的代码，当异常被抛出时，一个异常陷阱就自动发生，程序控制离开执行部分转入异常部分,一旦程序进入异常部分就不能再回到同一块的执行部分。下面是异常部分的一般语法：

```
EXCEPTION
  WHEN exception_name THEN
    Code for handing exception_name
  [WHEN another_exception THEN
    Code for handing another_exception]
  [WHEN others THEN
    code for handing any other exception.]
```

用户必须在独立的 WHEN 子串中为每个异常设计异常处理代码，WHEN OTHERS 子串必须放置在最后面作为缺省处理器处理没有显式处理的异常。当异常发生时，控制转到异常部分，ORACLE 查找当前异常相应的 WHEN..THEN 语句，捕捉异常，THEN 之后的代码被执行，如果错误陷阱代码只是退出相应的嵌套块，那么程序将继续执行内部块 END 后面的语句。如果没有找到相应的异常陷阱，那么将执行 WHEN OTHERS。在异常部分 WHEN 子串没有数量限制。

```
EXCEPTION

  WHEN inventory_too_low THEN
    order_rec.staus:='backordered';
    replenish_inventory(inventory_nbr=>
    inventory_rec.sku,min_amount=>order_rec.qty-inventory_rec.qty);
  WHEN discontinued_item THEN
    --code for discontinued_item processing
  WHEN zero_divide THEN
    --code for zero_divide
  WHEN OTHERS THEN
    --code for any other exception
END;
```

当异常抛出后，控制无条件转到异常部分，这就意味着控制不能回到异常发生的位置，当异常被处理和解决后，控制返回到上一层执行部分的下一条语句。

```
BEGIN
  DECLARE
    bad_credit;
  BEGIN
    RAISE bad_credit;
      --发生异常，控制转向；
  EXCEPTION
    WHEN bad_credit THEN
      dbms_output.put_line('bad_credit');
    END;

    --bad_credit 异常处理后，控制转到这里
  EXCEPTION
    WHEN OTHERS THEN
      --控制不会从 bad_credit 异常转到这里
      --因为 bad_credit 已被处理
END;
```

当异常发生时，在块的内部没有该异常处理器时，控制将转到或传播到上一层块的异常处理部分。

```
BEGIN
  DECLARE ---内部块开始
    bad_credit;
  BEGIN
    RAISE bad_credit;
      --发生异常，控制转向；
    EXCEPTION
    WHEN ZERO_DIVIDE THEN --不能处理 bad_credite 异常
      dbms_output.put_line('divide by zero error');
    END --结束内部块

      --控制不能到达这里，因为异常没有解决；
      --异常部分
```

```
    EXCEPTION
    WHEN OTHERS THEN
      --由于 bad_credit 没有解决，控制将转到这里
END;
```

异常传播

没有处理的异常将沿检测异常调用程序传播到外面，当异常被处理并解决或到达程序最外层传播停止。

在声明部分抛出的异常将控制转到上一层的异常部分。

```
BEGIN
executable statements
BEGIN
today DATE:='SYADATE'; --ERRROR
BEGIN --内部块开始
dbms_output.put_line('this line will not execute');
EXCEPTION
WHEN OTHERS THEN
--异常不会在这里处理
END;--内部块结束

EXCEPTION
WHEN OTHERS THEN
处理异常
END
```

执行部分抛出的异常将首先传递到同一块的异常部分，如果在同一块的异常部分没有处理这个异常的处理器，那么异常将会传播到上一层的异常部分中，一直到最外层。

在异常部分抛出的异常将控制转到上一层的异常部分。

处理异常将停止异常的传播和解决。有时用户希望在错误发生时，程序仍然能执行一些

动作，要达到这个目的，可以把希望执行的动作放在异常处理器中，然后执行不带参数的 RAISE 语句，RAISE 语句将重新抛出出现的异常，允许他继续传播。

```
DECLARE
order_too_old EXCEPTION;
BEGIN
RAISE order_too_old;
EXCEPTION
WHEN order_too_old THEN
DECLARE
file_handle UTL_FILE.FILE_TYPE;
BEGIN
--open file
file_handle:=UTL_FILE.FOPEN
(location=>'/ora01/app/oracle/admin/test/utlsir'
,filename=>'error.log'
.open_mode=>'W');
--write error stack
UTL_FILE.PUT_LINE(filehandle,
DBMS_UTILITY.FORMAT_ERROR_STACK);
--write the call stack
UTL_FILE.PUT_LINE(filehandle,
DBMS_UTILITY.FORMAT_CALL_STACK);
--close error log
UTL_FILE.FCLOSE(file_handle);
RAISE; --re-raise the exception
END
END
```

如果从 FORMAT_XXX_STACK 输出一个很大的值，那么使用 DBMS_OUTPUT 或 UTL_FILE 显示错误或调用堆的异常部分自身也会抛出异常，这两个堆常规下最多能返回 2000 字节，但 utl_file.put_line 被限制在 1000 字节以内，而 dbms_output.put_line 限制在 512 字节内。如果使用前面的代码并且不允许这种可能性，那么在异常处理器中将抛出一个未处理的异常。

GOTO 语句不能用于将控制从执行部分传递到异常部分或反之。

已命名异常

在 PL/SQL 块的异常部分只有已命名的异常才能被 WHEN 子串处理，ORACLE 包含了一系列已命名的异常，这些异常都声明在 STANDARD 包中，这些内建异常在这里就不一一讲述，有兴趣的读者可以查阅有关资料。