Oracle 数据库二

1.PL/SQL 简介

1.1 什么是 PL/SQL

PL/SQL 也是一种程序语言，叫做过程化 SQL 语言(Procedural Language/Structured Query Language)。是 Oracle 数据库对 SQL 语句的扩展。在普通 SQL 语句的使用上增加了编程语言的特点，所以 PL/SQL 就是把数据操作和查询语句组织在 PL/SQL 代码的过程性单元中，通过逻辑判断、循环等操作实现复杂的功能或者计算的程序语言。

1.2 为什么使用 PL/SQL

使用 PL/SQL 可以编写具有很多高级功能的程序，虽然通过多个 SQL 语句可能也能实现同样的功能，但是相比而言，PL/SQL 具有更为明显的一些优点：

1.能够使一组 SQL 语句的功能更具模块化程序特点；

2.采用了过程性语言控制程序的结构；

3.可以对程序中的错误进行自动处理，使程序能够在遇到错误的时候不会被中断；

4.具有较好的可移植性，可以移植到另一个 Oracle 数据库中；

5.集成在数据库中，调用更快；

6.减少了网络的交互，有助于提高程序性能

1.3 PL/SQL 语法

PL/SQL 是一种块结构的语言，这意味着 PL/SQL 程序包含一个或者多个逻辑块，每一

个逻辑块由三部分组成，其语法结构如下：

```
[declare]
    --声明部分，声明变量，类型，游标，以及局部存储过程和函数    可选
begin     --相当于 java 方法的左大括号
      --   执行部分，    过程，sql 语句
[exception]
     --异常处理部分，  异常处理   可选
end；       --结束相当于 java 方法的右大括号，  end 后面的分号必须写
```

例如使用 PL/SQL 块在控制台输出 helloWorld

```
begin
   dbms_output.put_line('helloWorld');   --输出到“输出”的控制台
end;
```

注意：如果在命令行执行 PL/SQL 块，并进行输出，需要打开 serveroutput,打开命令为：

```
set serveroutput on;
```

1.4 变量

1.4.1 变量定义:具有一定的存储空间，并且值可以改变的就叫变量;

一个变量只能存储一个值，只是这个值可以改变

我们可以在 declare 模块中定义变量，变量的定义语法：

```
变量名   数据类型[（长度）] [:=值]；
```

其中，变量名要求：

1） 只能是字母，数字，下划线，$,#

2）首字母必须是字母

3）长度不允许超过 30

4）不允许使用关键字 varchar

注意： 在 begin 中，不能定义新的变量。

1.4.2 变量的赋值

变量的赋值有下列两种方式：

1） 使用 变量名：=值，来给变量赋值

```
declare
--变量
 --第一种变量的赋值： 使用 :=
    v_char varchar2(20) :='helloWorld';
    v_num constant number :=100;


begin
    v_char:='aa';
    dbms_output.put_line(v_char);   --输出到“输出”的控制台
     dbms_output.put_line(v_num);
end;
```

直接赋值

2）使用 select 列名 into 变量名 通过sql查询赋值

编写一个 PL/SQL，可以通过员工编号查询员工姓名，并且使用变量保存员工姓名

```
declare
 v_name varchar2(20);
begin
--&字符串 ： 输入
  select ename into  v_name from emp where empno='&员工编号
';
  dbms_output.put_line('您查询的员工的姓名：'||v_name);
end;
```

练习： 编写一个通过员工编号，查询员工姓名，薪水的 PL/SQL

1.4.3 变量复杂的数据类型

1.%type

定义某个变量的数据类型与表的某个列的数据类型一样，使用方式：

```
变量名  表名.列名%type；
```

例如

```
declare
 v_name emp.ename%type;
 v_sal emp.sal%type;
begin
--&字符串 ： 输入
```

```
  select ename,sal into  v_name,v_sal from emp where empno='&员工编号';
  dbms_output.put_line('您查询的员工的姓名：'||v_name);
end;
```

2. %rowtype

%rowtype 用于定义不确定的类型的变量，%rowtype 可以理解成对数据库中表的某一行记录的一个拷贝，使用方式

```
  变量名  表名%rowtype；
```

例如：

```
  --查询某一个员工所有的信息
declare
 v_emp emp%rowtype;  --表示： v_emp 放的 emp 表中一行的数据
begin
--&字符串 ： 输入
  select * into  v_emp  from emp where empno='&员工编号';
  dbms_output.put_line(' 您 查 询 的 员 工 的 姓 名 ： '||v_emp.ename||' 薪 水 ：
'||v_emp.sal);
end;
```

1.5 流程语句

1.5.1 分支语句

1.5.1.1 IF 分支语句

If 语句的语法： else if → elsif

if和elsif的区别在于if会一直往下判断而elsif在成功判断一次之后就不会再往下进行判断了

```
if （条件） then
        pl/sql 或 sql 语句
    [elsif (条件) then ]
      ...可以有多个 elsif
     [else]
end if ;
```

示例：

```
--完成查看每个员工需要交的税的情况    大于等于 3000 交 1%     大于等于 1500 交
0.5%,低于 1500 的不要交税
declare
   v_sal emp.sal%type;
   v_rate emp.sal%type;
begin
  select sal into v_sal from emp where empno ='&id';
  if (v_sal = 3000) then
    v_rate := v_sal*0.01;
```

```
  elsif (v_sal =1500) then
    v_rate := v_sal*0.005;
   else
     v_rate:=0;
   end if;
   dbms_output.put_line(v_rate);
end;
```

1.5.2 循环语句

Oracle 的循环分为下列三种：

- Loop 循环　　使用最多
- While 循环
- For 循环

1.5.1.2 case 分支语句

当有多重情况时，也可以使用类似 java 的 switch 结构：case，语法结构如下：

```
case [变量名]
-when (条件)  then
when （条件） then
...
else
end case;
```

例如，查看员工的薪资情况，=3000 高工资， =1500 中等工资， <1500 低工资

```
declare
  v_sal emp.sal%type;
begin
  select sal into v_sal from emp where empno ='&id';
  case
      when v_sal=3000 then
       dbms_output.put_line('高工资');
      when v_sal=1000 then
        dbms_output.put_line('中等工资');
      else
          dbms_output.put_line('低工资');
  end case;
end;
```

1.5.2.1 Loop 循环

Loop 循环中，我们需要自定义循环结束的条件，语法如下：

```
  Loop
     Pl/SQL 语句  循环需要做的事情（循环体）
    exit  when 循环结束的条件;
  end loop ;
```

例如：

```
--输出 10 遍 helloWorld

declare

 i number :=0;

begin

  loop

       i:= i+1;

       dbms_output.put_line('helloWorld');

       exit when i=10;

  end loop;

end;
```

练习：使用 Loop 循环，计算 10 的阶乘

1.5.2.2 while 循环

While 循环与 java 中的 while 相似，它的语法：

```
while (布尔表达式) loop

            --sql 语句；

 end loop;
```

例如：

```
declare

 i number :=0;

begin

  while i<10 loop
```

```
        dbms_output.put_line('helloWorld');
        i:= i+1;
   end loop;
end;
```

1.5.3 for 循环

For 循环的语法结构：

```
for i in 最小值..最大值 loop
            sql 语句
 end loop;
```

注意：i 不需要定义

```
begin
   for i in 1..10 loop
      dbms_output.put_line(i);
   end loop;
end;
```

逆序打印：使用 reverse 关键字：如下

```
begin
   for i in   reverse 1..10 loop
      dbms_output.put_line(i);
   end loop;
end;
```

1.6 异常处理

1.6.1 预定义异常

对于 PL/SQL 过程中的语句，也有类似 java 中的异常处理，对于异常，Oracle 中也提供了一部分预定义的异常（也就是说，oracle 可以直接捕获的异常，并且提供异常名），预定义的异常如下：

预定义异常	说明
NO_DATA_FOUND	语句无返回数据
TOO_MANY_ROWS	在执行 select into 语句时返回多行时出现
LOGIN_DENIED	使用无效的用户名和口令登录 Oracle
INVALID_NUMBER	试图将一个非有效的字符串转换成数字
DUP_VAL_ON_INDEX	重复的值存储在使用唯一索引的列中
ACCESS_INTO_NULL	试图给一个没有初始化的对象赋值
ZERO_DIVIDE	除以 0
VALUE_ERROR	算术或转换错误
TIMEOUT_ON_RESOURCE	在等待资源时发生超时

对预定义的异常进行处理：

```
declare
  v_name emp.ename%type;
begin
```

```
 select ename into v_name from emp where empno='&id';
dbms_output.put_line(v_name);
exception   --异常的处理
   when NO_DATA_FOUND then
      dbms_output.put_line('未找到数据');
   when others then
       dbms_output.put_line('系统忙，请稍后再试');
end;
```

1.6.2 非预定义异常

Oracle 数据库中还定义了一些异常，但是没有异常名，只有异常代码，这类异常，我们称之为非预定异常,比如说,违反外键约束的异常,就没异常名,但是有异常代码:-2291，那我们怎么处理呢？

我们处理的思路如下：

1. 申明异常，就是给这个非预定的异常取一个名，语法：

   异常名 exception;

2. 把申明的异常，绑定到 oracle 提供的非预定的异常上，也就是绑定对应的异常代码，语法如下：

   PRAGMA EXCEPTION_INIT(自定义异常名，异常代码)；

3. 在 PL/SQL 块中处理异常

   示例：

```
declare
```

```
        --1.声明一个变量
        myExe exception;
        PRAGMA EXCEPTION_INIT(myExe,-2291);
    begin
        insert into   tb_users values(5,'小红',22,4);
        exception
          when myExe then
            dbms_output.put_line('系统忙，请稍后再试');
end;
```

1.6.3 自定义异常

有些异常是oracle 没有定义的异常，我们可能根据实际需求，需要自定义一个异常，比方说，我们根据业务需求，需要制定一个奖金为 null 的异常，然后在我们的 PL/SQL 块中可以处理这个异常，那我们处理的步骤：

1. 申明一个异常， 异常名 exception;
2. 在指定的地方抛出这个异常 RAISE 异常名；
3. 捕获这个异常，然后进行处理

示例：

```
declare
 v_comm emp.comm%type;
 comm._is_null   exception;
```

```
begin
  select comm into v_comm from emp where empno=7654;
  if v_comm is null then
    RAISE comm._is_null;
  else
   dbms_output.put_line(v_comm);
  end if;
   exception
     when comm._is_null then
       dbms_output.put_line('奖金为空');
end;
```

2.游标 逐行处理查询结果，以编程的方式访问数据

游标是 SQL 的一个内存工作区，由系统或用户以变量的形式定义。游标的作用就是用于临时存储从数据库中提取的数据块。在某些情况下，需要把数据从存放在磁盘的表中调到计算机内存中进行处理，最后将处理结果显示出来或最终写回数据库。这样数据处理的速度才会提高，否则频繁的磁盘数据交换会降低效率。 Redis 内存穿透

游标有两种类型：显式游标和隐式游标。在前述程序中用到的 SELECT...INTO...查询语句，一次只能从数据库中提取一行数据，对于这种形式的查询和 DML 操作，系统都会使用一个隐式游标。但是如果要提取多行数据，就要由程序员定义一个显式游标，并通过与游标有关的语句进行处理。显式游标对应一个返回结果为多行多列的 SELECT 语句。

游标一旦打开，数据就从数据库中传送到游标变量中，然后应用程序再从游标变量中分

解出需要的数据，并进行处理。

2.1 隐式游标

如前所述，DML 操作和单行 SELECT 语句会使用隐式游标，它们是：

* 插入操作：INSERT。

* 更新操作：UPDATE。

* 删除操作：DELETE。

* 单行查询操作：SELECT ... INTO ...。

当系统使用一个隐式游标时，可以通过隐式游标的属性来了解操作的状态和结果，进而控制程序的流程。隐式游标可以使用名字 SQL 来访问，但要注意，通过 SQL 游标名总是只能访问前一个 DML 操作或单行 SELECT 操作的游标属性。所以通常在刚刚执行完操作之后，立即使用 SQL 游标名来访问属性。游标的属性有四种，如下所示。

隐式游标的属性	返回值类型	说明
SQL%ROWCOUNT	整型	代表 DML 语句成功执行的行数
SQL%FOUND	布尔型	值为 true 表示插入，删除，更新，和单行查询操作成功
SQL%NOTFOUND	布尔型	与 SQL%FOUND 相反
SQL%ISOPEN	布尔型	DML 执行过程中为真，结束为 false

使用隐式游标的属性，判断对单行查询是否成功。

```
DECLARE
   v_emp  emp%ROWTYPE;
BEGIN
```

```
  SELECT * INTO v_emp FROM emp WHERE empno = '&id';
  IF SQL%FOUND THEN
    dbms_output.put_line('查询到数据,员工姓名:
'||v_emp.ename);
   ELSE
     dbms_output.put_line('没有查询到数据');
   END IF;
END;
```

2.2 显示游标

当查询到多行数据的时候，需要使用显示游标来进行操作。显示游标的使用分为四个步骤：

1. 声明游标

   在 Declear 部分按以下格式声明游标

这里的参数是否也分成in和out参数？

```
CURSOR 游标名[(参数名 1  数据类型 1[,参数名 2  数据类型 2...])]
IS  SELECT 子句；
```

注意：参数是可选部分，所定义的参数可以出现在 SELECT 语句的 WHERE 子句中，如果定义了参数，则必须在打开游标时，传递相应的数值。

这句话是不是代表游标参数的in属性？

2. 打开游标

   在可执行部分（begin），按照如下语法打开游标：

```
OPEN 游标名[(参数值 1[,参数值 2...])]
```

在打开游标时，SELECT 语句查询到的结果就被传递到游标工作区。

3. 提取数据

在可执行部分，按照如下语法将游标工作区中的数据提取到变量中。提取操作必须在打开游标之后进行。

FETCH 游标名 INTO 变量名 1【，变量名 2....】
FETCH 游标名 INTO 记录变量

游标打开后有一个指针指向数据区，FETCH 语句一次返回指针所指向的一行数据，要返回多行需要重复执行，可以使用循环语句来实现，控制循环可以通过判断游标的属性来进行。游标的属性如下：

属性名	**说明**
%found	用于检验游标是否成功，通常在 FETCH 语句前使用，当游标按照条件查询出一条记录时，返回 true
%isopen	判断游标是否处于打开状态，试图打开一个已经打开或者已经关闭的游标，将会出现错误
%notfound	与%found 的作用相反，当按照条件无法查询到记录时，返回 true
%rowcount	循环执行游标读取数据时，返回检索出的记录数据的行数

使用变量名来接收从游标中的数据，需要要事先定义，变量的个数和类型应与 SELECT 语句中的字段变量的个数和类型一致。

我们也可以将一行数据读取到记录变量中，需要使用%rowtype 事先定义记录变量，这种方式使用起来比较简单，不需要定义多个变量。

4. 关闭游标

语法如下：

```
CLOSE 游标名；
```

显示游标打开后，必须显示地关闭。游标一旦关闭，游标占用的资源就会被释放，游标变成无效，必须重新打开才能使用。

示例 1：

使用游标提取 emp 表中 7788 的姓名和岗位：

```
DECLARE
    v_ename VARCHAR2(10);
    v_job VARCHAR2(10);
    --声明游标
    CURSOR emp_cursor IS
    SELECT ename,job FROM emp WHERE empno=7788;
BEGIN
    --打开游标
    OPEN emp_cursor;
    --提取数据
    FETCH emp_cursor INTO v_ename,v_job;
        DBMS_OUTPUT.PUT_LINE(v_ename||','||v_job);
    --关闭游标
    CLOSE emp_cursor;
END;
```

静态游标可以提取单独几列数据

示例 2，使用游标查询某个部门的所有员工信息：

```
DECLARE
```

```
      v_emp emp%Rowtype;
      v_deptno emp.deptno%TYPE := &deptno;
      --声明游标
      CURSOR emp_cursor IS
      SELECT * FROM emp WHERE deptno=v_deptno;
BEGIN
     --打开游标
     OPEN emp_cursor;
     --循环提取数据
     LOOP
        FETCH emp_cursor INTO v_emp;
        EXIT WHEN emp_cursor%NOTFOUND;

DBMS_OUTPUT.PUT_LINE(v_emp.empno||','||v_emp.ename||'
,'||v_emp.sal);
      END LOOP;
     --关闭游标
     CLOSE emp_cursor;
END;
```

示例 3 通过游标更新 薪水低于 3000 的雇员的薪水增加 500 元 ,增加后超过 3000 元，则薪水为 3000.

先备份数据：

```
CREATE TABLE emp_bak AS
 SELECT * FROM   emp
```

```
  DECLARE
    v_emp emp_bak%ROWTYPE; --员工表行记录变量
    CURSOR emp_cursor IS SELECT * FROM emp_bak WHERE sal<3000;
-- 找到sal小于3000员工绑定游标
 BEGIN
      OPEN emp_cursor ;--打开游标
      LOOP
         FETCH emp_cursor INTO v_emp; --提取游标中的记录
         EXIT WHEN emp_cursor%NOTFOUND;
         IF v_emp.sal+5003000 THEN
            UPDATE emp_bak SET sal=3000 WHERE
empno=v_emp.empno;
         ELSE
            UPDATE emp_bak SET sal=sal+500 WHERE
empno=v_emp.empno;
          END IF;
        END LOOP;
        COMMIT;
        dbms_output.put_line('少于3000的员工工资已加');
```

```
    CLOSE emp_cursor;
END;
```

2.3 动态游标

在前面我们已经学习了隐式游标和显示游标，我们把隐式游标和显示游标都称为静态游标，因为在使用它们之前，游标的定义就已经完成了，不能再更改。而在实际开发中，可能在声明游标时，我们无法确定该游标绑定的 select 是哪个，或者定义的这个游标可以根据业务需求，进行动态的更改 select 语句，那此时，我们就需要使用到动态游标，也叫做参考游标。

动态游标是指在声明时还没有设定 select，而是在打开游标时，可以对其进行修改。而动态游标又分为强类型游标和弱类型游标。

强类型动态游标是指在声明变量时使用 return 关键字定义游标的返回值类型

弱类型动态游标是指在声明变量时不使用 return 关键字定义游标的返回类型

一般动态游标有 REF CURSOR、REF CURSOR RETURN、SYS_REFCURSOR。

REF CURSOR RETURN 为强类型，REF CURSOR 为弱类型、SYS_REFCURSOR 为系统预定义（在 STANDARD 包中定义的）弱类型。

示例：使用弱类型动态游标，完成查询所有的部门信息，再使用该游标查询部门编号为 10 的员工信息

```
DECLARE
    --第一步：定义一个动态游标类型
  TYPE  c_cursor IS  REF  CURSOR ;
   --第二步：定义一个动态游标变量
```

```
  v_ref_cur c_cursor; 意味着这个变量其实就是一个动态游标
  v_emprow  emp%ROWTYPE ;
  v_deptrow  dept%ROWTYPE ;
BEGIN
  OPEN  v_ref_cur FOR     在打开这个游标时才对其进行赋值
    SELECT  * from dept;
  LOOP
    FETCH  v_ref_cur
      INTO  v_deptrow;
    EXIT  WHEN  v_ref_cur%NOTFOUND;
    dbms_output.put_line(v_deptrow.dname);
  END  LOOP ;
  CLOSE  v_ref_cur;

  dbms_output.put_line('-------------------------');
  OPEN  v_ref_cur for
    SELECT  * FROM  emp WHERE  deptno = 10;
  LOOP
    FETCH  v_ref_cur
      INTO  v_emprow;
    EXIT  WHEN  v_ref_cur%NOTFOUND ;
    dbms_output.put_line(v_emprow.ename);
```

```
  END  LOOP;
  CLOSE  v_ref_cur;

EXCEPTION
  WHEN  OTHERS  THEN
    dbms_output.put_line(sqlerrm);
END;
```

系统预定义弱类型动态游标（SYS_REFCURSOR），可以省略动态游标的定义。

```
DECLARE
  --定义一个系统预定义的动态游标
  v_ref_cur SYS_REFCURSOR;
  v_emprow  emp%ROWTYPE ;
  v_deptrow  dept%ROWTYPE ;
BEGIN

  OPEN  v_ref_cur FOR
    SELECT  * from dept;
  LOOP
    FETCH  v_ref_cur
      INTO  v_deptrow;
    EXIT  WHEN  v_ref_cur%NOTFOUND;
    dbms_output.put_line(v_deptrow.dname);
```

```
  END  LOOP ;
  CLOSE  v_ref_cur;

  dbms_output.put_line('--------------------------');
  OPEN  v_ref_cur for
    SELECT  * FROM  emp WHERE  deptno = 10;
  LOOP
    FETCH  v_ref_cur
      INTO  v_emprow;
    EXIT  WHEN  v_ref_cur%NOTFOUND ;
    dbms_output.put_line(v_emprow.ename);
  END  LOOP;
  CLOSE  v_ref_cur;

EXCEPTION
  WHEN  OTHERS  THEN
    dbms_output.put_line(sqlerrm);
END;
```

3. 存储过程

所谓的存储过程（Stored Procedure），就是一组用于完成特定数据库功能的 SQL 语句集，该 SQL 语句集经过编译后存储在数据库系统中，在使用的时候，用户通过调用指定已

经定义好的存储过程并执行它，从而完成一系列的数据库操作。

3.1 存储过程的语法

```
create [or replace] procedure  存储过程名
        [(参数 1 model  类型 1，参数 2   model  类型 2,.....)]
        is/as
        [变量。游标声明]
        begin   --相当于 java 中的{
                --执行语句
        [exception
                --异常处理
        ]
   end;   --相当于 java 的 }
```

语法说明： model 的取值有如下三种取值：

- in 输入参数，不能再在过程中改变
- out 输出参数， 就是把结果输出，这个参数必须在过程中被赋值
- in out 输入输出参数，只用于参数的交换，很少用到

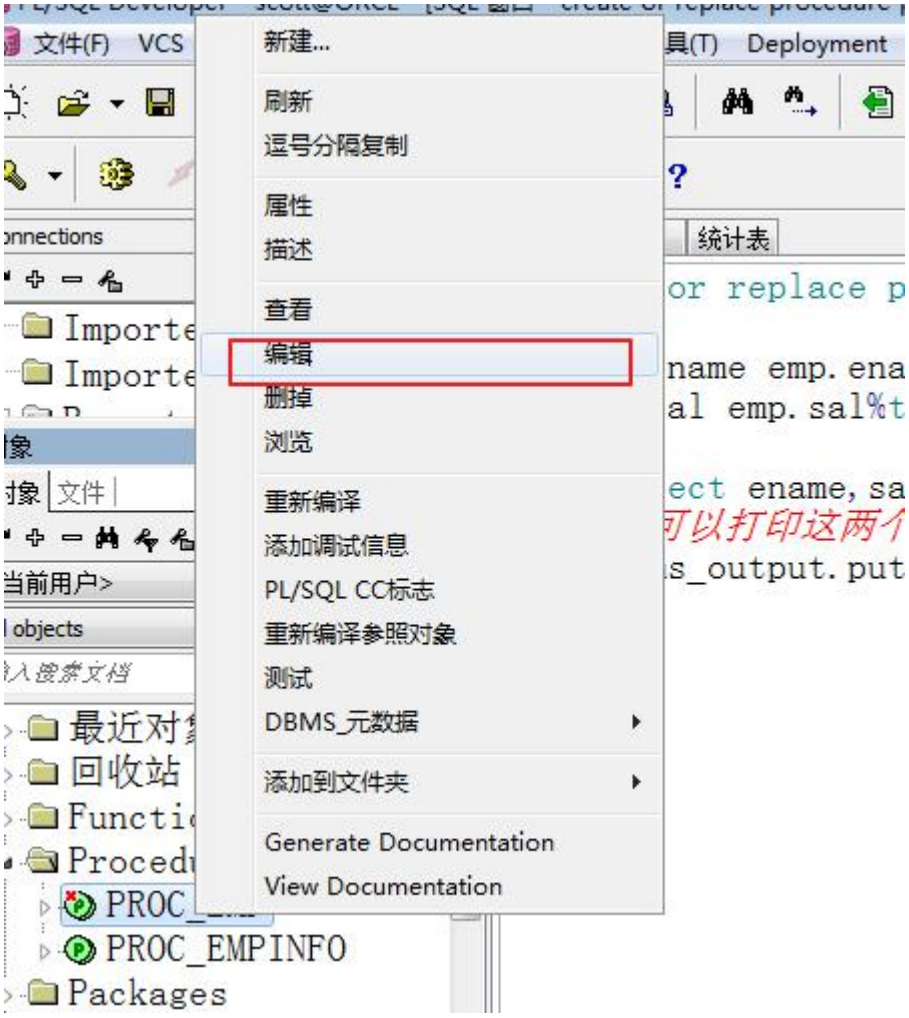
示例：简单的存储过程，不带参数

```
create or replace procedure proc_emp
is
    v_ename emp.ename%type;
    v_sal emp.sal%type;
```

```
begin
    select ename,sal into v_ename, v_sal from emp   where empno = 7788;
    --可以打印这两个变量
    dbms_output.put_line(v_ename || ', ' || v_sal);
end;
```

注意： 我们在 PL/SQL 中，写完存储过程，一定要进行编译，并且要查看存储过程是否编译有错误。查看编写的过程是否有错，点击 procedures 文件夹，查看你编写的过程上是否有小红叉，如下图：

如果有小红叉，则说明有错误，选择存储过程，右键，再选择“编辑“(edit)，可以在存储过程的编辑界面查看错误的地方，修改，再进行编译，一定要编译成功的存储过程才能被调用。



3.2 存储过程的执行

第一种： 在 DOS 小黑屏上或者 PL/SQL Developer 的命令窗口执行

执行的语法：

execute

```
exec  存储过程名；
```

注意：如果存储过程中包含 dbms_output 这个输出语句时，需要进行设置：

```
set  serverout on；
```

```
SQL> set serverout on;
SQL> exec proc_emp;

SCOTT, 3000

PL/SQL procedure successfully completed

SQL>
```

第二种： 在 PL/SQL Developer 的 SQL 窗口执行：

```
begin

    存储过程；

end；
```

```
BEGIN
  proc_emp;
END;
```

3.3 存储过程的参数模式

- in 输入参数， 可以在定义参数时，省略不写。存储过程参数默认的模式是 in

  例如：给某个部门加薪，最多一次加薪不能超出 1000

  我们先把 emp 表备份一份，使用如下 SQL 语句：

  ```
  create table emp2 as select * from emp;
  ```

  编写存储过程：

```
create or replace procedure proc_addSal
  (p_deptno in emp2.deptno%type, p_money   emp2.sal%type)
  is


  begin
        if (p_money   1000) then
            dbms_output.put_line('金额超出 1000，请与经理联系');
            return;   --终止过程往下执行，直接跳出过程
        else --可以加薪
            update emp2 set sal = sal + P_money   where deptno =
P_deptno;
            commit;
        end if;
  end;
```

执行带参数的存储过程：

```
begin
  proc_addSal(10,1200);
end;
```

```
--参数赋值：参数名 = 值
begin
   proc_addSal(p_money=100,p_deptno = 20);
```

```
end;
```

- out 输出参数

示例： 根据员工编号查找相应的员工姓名，如果有，则显示员工姓名，如果没有，则显示“未找到员工编号为 XXXX 的员工”

```
create or replace procedure proc_selectEmp
  (p_empno emp2.empno%type, p_ename out  emp2.ename%type)
                in？
is



begin
   select ename into p_ename from emp2 where empno = p_empno;
exception
   when no_data_found then
        dbms_output.put_line('未找到员工编号为'||p_empno||'的员工');
end;


--执行，带输出参数过程，必须声明一个变量来接收输出参数
declare
   v_ename emp2.ename%type;
begin      procedure中out参数中的值需要用变量来赋值,
           因此此处只能输入变量，否则无法赋值
   proc_selectEmp('&empno',v_ename);
   dbms_output.put_line(v_ename);
end;
```

执行带输出参数过程，必须声明变量来接收输出参数：

```
declare
    v_ename emp2.ename%type;
  begin
    proc_selectEmp('&empno',v_ename);
    dbms_output.put_line(v_ename);
  end;
```

in和out参数的作用好像是带有输入和输出功能的存储元件，如果是多行多列，这个参数就需要使用游标来代替

如果存储过程执行查询返回的是多行记录，则需要使用游标

示例：查询指定部门的编号的员工信息

```
 create or replace procedure pro_getEmpByDeptno(v_deptno IN
emp.deptno%TYPE,cursor_emp out sys_refcursor)
     is
     begin
        open cursor_emp for select * from emp_bak where
deptno=v_deptno;
end;
```

执行：

```
declare
      v_emp emp%rowtype;
      cursor_emp  sys_refcursor;
```

```
      begin
         pro_getEmpByDeptno(&deptno,cursor_emp);
         loop
           fetch cursor_emp into v_emp;
            exit when cursor_emp%NOTFOUND;
            dbms_output.put_line(v_emp.deptno||'
'||v_emp.empno||' '||v_emp.ename||' '||v_emp.sal);
           end loop;
           close cursor_emp;
       end;
```

3.4 存储过程授权

我们可以把某个用户创建的存储过程授予给其他用户来执行。

```
grant  execute on 存储过程名  to 用户名;
```

例如：

```
grant  execute on pro_getEmpByDeptno  to  suke;
```

3.5 删除存储过程

```
drop procedure 存储过程名;
```

例如：

```
drop procedure getEmpByDeptno  ;
```

4.函数

Oracle 的函数是用于返回特定数据，函数的语法如下：

```
create [or replace] function  函数名
        (参数 1  类型 1，  参数 2  类型 2,....)  也可以使用out参数
        return  返回值类型
        is/as
            [定义变量]
        begin
                --执行语句
                return  结果；
        [exception
                --异常的处理
        ]
 End；
```

语法说明：

1. 函数的参数只能是 in 模式，可以省略 in
2. 函数声明的时候，必须使用 return 加返回值类型，注意，return 的返回值类型只需要告诉类型，不需要定义长度，例如： return varchar2；
3. 函数的结果必须通过 return 返回出去，也就是说，在 begin 中使用 return 结果；

4.1 无参函数：

编写一个函数，让它返回“helloworld”

```
create or replace function fun_hello
 return varchar2     --没有长度，也不需要加“;”
 is
 begin
        return 'helloWord';
 end;
```

函数的执行，函数只能配合其他 sql 语句一起执行，比如 select，dbms

```
select fun_hello() from dual;
```

```
begin
        dbms_output.put_line(fun_hello());
end;
```

4.2 有参函数

编写一个函数，计算某个员工的年收入（sal+comm）*12

```
create or replace function fun_yearSal
 (p_empno emp.empno%type)
 return emp2.sal%type  不加分号
 is
```

```
    v_sal emp2.sal%type;
    v_comm emp2.comm%type;
    v_yearSal emp2.sal%type;
 begin
      select sal,comm into v_sal, v_comm from emp2 where empno =
P_empno;
      v_yearSal := (v_sal+nvl(v_comm,0))*12;
      return v_yearSal;
 end;
```

```
--执行
declare
  v_empno emp2.empno%type;
begin
   v_empno := '&empno';
   dbms_output.put_line(fun_yearSal(v_empno));
end;
```

4.3 存储过程与函数的区别

1.关键字不一样,过程使用 procedure,函数使用 function

2.在过程中，所有的参数都放在小括号里面，通过 in 来表示传入的参数，通过 out 来表示返回的数据;

在函数中，小括号中放的都是传入的参数，也没有 in/out 的写法，返回的数据都通过

return 来完成

3.在过程中，对于需要返回的结果，只需要存放到 out 修饰的参数里面就可以了

在函数中，过程语句执行完之后，必须通过 return 返回执行结果

4.调用过程时，如果有返回数据，需要把变量放到过程的小括号中来接收结果；

调用函数的时候，则通过给变量赋值的方式来接收函数执行的结果

5.过程一般供程序调用，而函数一般由其他过程或者函数来调用

5.Oracle 其他对象

5.1 序列

Oracle 中在增加一条数据时，为了实现自增长，需要使用序列。序列的语法：

```
create sequence  序列名
        [INCREMENT BY n]            //每次增长值
         [START WITH n]                  //起始值
        [{MAXVALUE n| NOMAXVALUE}]          //最大值
        [{MINVALUE n| NOMINVALUE}]      //最小值
```

示例：

```
create sequence myseq
 minvalue 1
 maxvalue 1000
 start with 1
```

```
increment by 1

cache 20;
```

序列的使用：

1.nextval，让序列产生一个数值

```
select myseq.nextval from dual;
```

2.currval 当前正在使用的序列值（不会让序列产生值），必须在序列使用（nextval）之后才有

```
select myseq.currval from dual;
```

3.删除序列，删除序列不会删除序列已产生的值

```
drop sequence myseq;
```

5.2 包

在一个大型项目中，可能有很多模块，而每个模块又有自己的过程、函数等。而这些过程、函数默认是放在一起的（如在 PL/SQL 中，过程默认都是放在一起 的，即 Procedures 中），这些非常不方便查询和维护，甚至会发生误删除的事件。PL/SQL 为了满足程序模块化的需要，引入了包的构造。通过使用包就可以分类管理过程和函数等。

（1）包是一种数据库对象，相当于一个容器。将逻辑上相关的过程、函数、变量、常量和游标组合成一个更大的单位。用户可以从其他 PL/SQL 块中对其进行引用

（2）包类似于 C++和 JAVA 语言中的类，其中变量相当于类中的成员变量，过程和函数相当于类方法。把相关的模块归类成为包，可使开发人员利用面向对象的方法进行开发，具有面向对象程序设计语言的特点，

（3）PL/SQL 的包具有信息隐蔽性(information hiding)，仅在算法和数据结构设计有关

层可见。可将过程说明和过程体组成一个程序单位。也可将过程说明与它的过程体分开。也可在包中定义过程,而该过程在包说明中没有定义过程说明,这样定义的过程仅在包内使用。

（4）在 PL/SQL 程序设计中，使用包不仅可以使程序设计模块化，对外隐藏包内所使用的信息（通过使用私用变量），而写可以提高程序的执行效率。因为，当程序首次调用包内函数或过程时，ORACLE 将整个包调入内存，当再次访问包内元素时，ORACLE 直接从内存中读取，而不需要进行磁盘 I/O 操作，从而使程序执行效率得到提高

一个包由两个分开的部分组成，包的定义和包的实现，语法如下：

包的定义语法：

```
create or replace package  包名
is
   过程或函数的定义
   ....
end;
```

示例：

```
 create or replace package mypkg
 is
        procedure findSalByEmpno(v_no in emp.empno%type,v_sal out emp.sal%type);
         procedure findEmpByEmpno(v_no in emp.empno%type,empinfo out emp%rowtype);
         function findSalByEmpnoFun(v_no emp.empno%type) return emp.sal%type;
```

```
        function findEmpByEmpnoFun(v_no emp.empno%type) return
emp%rowtype;
 end;
```

这里分别创建了两个过程和两个函数，分别用来根据员工编号获取员工工资或员工信息。

接下来需要创建包的主体（实现包里面的过程和函数）：

```
--实现时，名字必须提前定义过
--必须加一个 body 关键字，表示这个是实现
 create or replace package body mypkg
 is
     procedure findSalByEmpno(v_no in emp.empno%type,v_sal out
emp.sal%type)
          is
          begin
            select sal into v_sal from emp where empno=v_no;
          end;

        procedure findEmpByEmpno(v_no in emp.empno%type,empinfo out
emp%rowtype)
          is
              cursor emp_info is select * from emp where empno=v_no;
          begin
              open emp_info;
```

函数与函数以及存储过程之间可以互相调用，包括不同包之间（加包名）

全局变量定义的位置　　同一范围内不允许有同名变量

局部变量（优先使用）定义的位置

```
            fetch emp_info into empinfo;
            close emp_info;
        end;

    function findSalByEmpnoFun(v_no emp.empno%type) return
emp.sal%type
        is
            v_sal emp.sal%type;
        begin
          select sal into v_sal from emp where empno=v_no;
          return v_sal;
        end;

    function findEmpByEmpnoFun(v_no emp.empno%type) return
emp%rowtype
        is
            cursor emp_info is select * from emp where empno=v_no;
            empinfo emp%rowtype;
        begin
             open emp_info;
             fetch emp_info into empinfo;
             close emp_info;
```

```
                return empinfo;

        end;

end;
```

包实现以后，包中的过程和函数都可以调用了：

```
--包中的过程调用

declare

        v_id emp.empno%type;

        v_emp emp%rowtype;

        v_sal emp.sal%type;

 begin

    v_id := &员工编号;

    mypkg.findSalByEmpno(v_id,v_sal);

    dbms_output.put_line(v_sal);

    mypkg.findEmpByEmpno(v_id,v_emp);

    dbms_output.put_line('编号 :'||v_emp.empno||',名字 :'||v_emp.ename||',职位 :
'||v_emp.job);

 end;
```

```
--包中的函数调用：

declare

        v_id emp.empno%type;

        v_emp emp%rowtype;

        v_sal emp.sal%type;
```

```
begin
        v_id := &员工编号;
        v_sal := mypkg.findSalByEmpnoFun(v_id);
        dbms_output.put_line(v_sal);
        v_emp := mypkg.findEmpByEmpnoFun(v_id);
        dbms_output.put_line('编号 :'||v_emp.empno||',名字 :'||v_emp.ename||',
职位 : '||v_emp.job);
end;
```

用来放return的值

用来输出值

5.3 视图

5.3.1 什么是视图

视图(view)，也称虚表，不占用物理空间，这个也是相对概念，因为视图本身的定义语句还是要存储在数据字典里的。视图只有逻辑定义。每次使用的时候,只是重新执行 SQL。

视图是从一个或多个实际表中获得的，这些表的数据存放在数据库中。那些用于产生视图的表叫做该视图的基表（原表）。一个视图也可以从另一个视图中产生。

视图的定义存在数据库中，与此定义相关的数据并没有再存一份于数据库中。通过视图看到的数据存放在基表中。

视图看上去非常象数据库的物理表，对它的操作同任何其它的表一样。当通过视图修改数据时，实际上是在改变基表中的数据；相反地，基表数据的改变也会自动反映在由基表产生的视图中。由于逻辑上的原因，有些 Oracle 视图可以修改对应的基表，有些则不能（仅仅能查询）。

5.3.2 视图的作用

1)提供各种数据表现形式，可以使用各种不同的方式将基表的数据展现在用户面前，以便符合用户的使用习惯(主要手段: 使用别名)；

2)隐藏数据的逻辑复杂性并简化查询语句，多表查询语句一般是比较复杂的，而且用户需要了解表之间的关系，否则容易写错；如果基于这样的查询语句创建一个视图，用户就可以直接对这个视图进行"简单查询"而获得结果．这样就隐藏了数据的复杂性并简化了查询语句.这也是 oracle 提供各种"数据字典视图"的原因之一,all_constraints 就是一个含有 2 个子查询并连接了 9 个表的视图(在 catalog.sql 中定义)；

3)执行某些必须使用视图的查询．某些查询必须借助视图的帮助才能完成．比如，有些查询需要连接一个分组统计后的表和另一表，这时就可以先基于分组统计的结果创建一个视图，然后在查询中连接这个视图和另一个表就可以了；

4)提供某些安全性保证．视图提供了一种可以控制的方式，即可以让不同的用户看见不同的列，而不允许访问那些敏感的列，这样就可以保证敏感数据不被用户看见；

5)简化用户权限的管理．可以将视图的权限授予用户，而不必将基表中某些列的权限授予用户，这样就简化了用户权限的定义。

5.3.3 创建视图

要在当前方案中创建视图，用户必须具有 create view 系统权限；要在其他方案中创建视图，用户必须具有 create any view 系统权限．视图的功能取决于视图拥有者的权限．

```
create [ or replace ] [ force ] view [schema.]view_name
 as
 select ...
 [ with check option ]
 [ with read only ];
```

其中，or replace：如果存在同名的视图，则使用新视图"替代"已有的视图

force："强制"创建视图,不考虑基表是否存在,也不考虑是否具有使用基表的权限

with check option：指定对视图执行的 dml 操作必须满足“视图子查询”的条件即,对通过视图进行的增删改操作进行"检查",要求增删改操作的数据，必须是 select 查询所能查询到的数据,否则不允许操作并返回错误提示．默认情况下，在增删改之前"并不会检查"这些行是否能被 select 查询检索到

with read only：创建的视图只能用于查询数据，而不能用于更改数据.

简单视图创建如下：

```
create or replace view empdept as
 select
e.empno,e.ename,e.job,e.mgr,e.hiredate,e.sal,e.comm,e.deptno,d.dname,d.loc
 from emp e,dept d
 where e.deptno = d.deptno
 with read only;
```

6.事务

为了方便演示事务，我们需要创建一个 account 表：

```
CREATE TABLE account(
    id NUMBER PRIMARY KEY,
    NAME VARCHAR2(30),
    balance NUMBER（10,2）
);
CREATE SEQUENCE seq_account;

INSERT INTO account(id,NAME,balance) VALUES(seq_account.nextval,'zs', 100000);
INSERT INTO account(id,NAME,balance) VALUES(seq_account.nextval,'ls', 100000);
INSERT INTO account(id,NAME,balance) VALUES(seq_account.nextval,'ww', 100000);
commit;
SELECT * FROM account;
```

6.1 什么是事务

银行转账！张三转 10000 块到李四的账户，这其实需要两条 SQL 语句：

- 给张三的账户减去 10000 元；
- 给李四的账户加上 10000 元。

如果在第一条 SQL 语句执行成功后，在执行第二条 SQL 语句之前，程序被中断了（可能是抛出了某个异常，也可能是其他什么原因），那么李四的账户没有加上 10000 元，而张三却减去了 10000 元。这肯定是不行的！

你现在可能已经知道什么是事务了吧！**事务中的多个操作，要么完全成功，要么完全失败！不可能存在成功一半的情况！**也就是说给张三的账户减去 10000 元如果成功了，那么给李四的账户加上 10000 元的操作也必须是成功的；否则给张三减去 10000 元，以及给李四加上 10000 元都是失败的！

6.2 事务的四大特性（ACID）

事务的四大特性是：

- 原子性（Atomicity）：事务中所有操作是不可再分割的原子单位。事务中所有操作要么全部执行成功，要么全部执行失败。
- 一致性（Consistency）：事务执行后，数据库状态与其它业务规则保持一致。如转账业务，无论事务执行成功与否，参与转账的两个账号余额之和应该是不变的。
- 隔离性（Isolation）：隔离性是指在并发操作中，不同事务之间应该隔离开来，使每个并发中的事务不会相互干扰。
- 持久性（Durability）：一旦事务提交成功，事务中所有的数据操作都必须被持久化到数据库中，即使提交事务后，数据库马上崩溃，在数据库重启时，也必须能保证通过某种机制恢复数据。

6.3 Oracle 中的事务

在 Oracle 数据库中，当我们自行了增，删，改操作之后，都必须提交事务。事务的开始： begin，事务的结束：commit 或者 rollback。

在执行 SQL 语句之前，或者 begin，这就开启了一个事务（事务的起点），然后可以执行多条 SQL 语句，最后要结束事务，commit 表示提交，即事务中的多条 SQL 语句所做出

的影响会持久化到数据库中。或者 rollback，表示回滚，即回滚到事务的起点，之前做的所有操作都被撤消了。

下面演示 zs 给 ls 转账 5000 元的示例

```
BEGIN
  UPDATE account SET balance=balance-10000 WHERE id=1;
  UPDATE account SET balance=balance+10000 WHERE id=2;
  ROLLBACK;
END;
```

```
BEGIN
  UPDATE account SET balance=balance-10000 WHERE id=1;
  UPDATE account SET balance=balance+10000 WHERE id=2;
  COMMIT;
END;
```