Oracle第三天

# 索引INDEX

## 索引的概念特性和作用

概念:

简单的说，相当于一本书的目录。(数据库中的索引相当于字典的目录(索引)),它的作用就是提升查询效率。

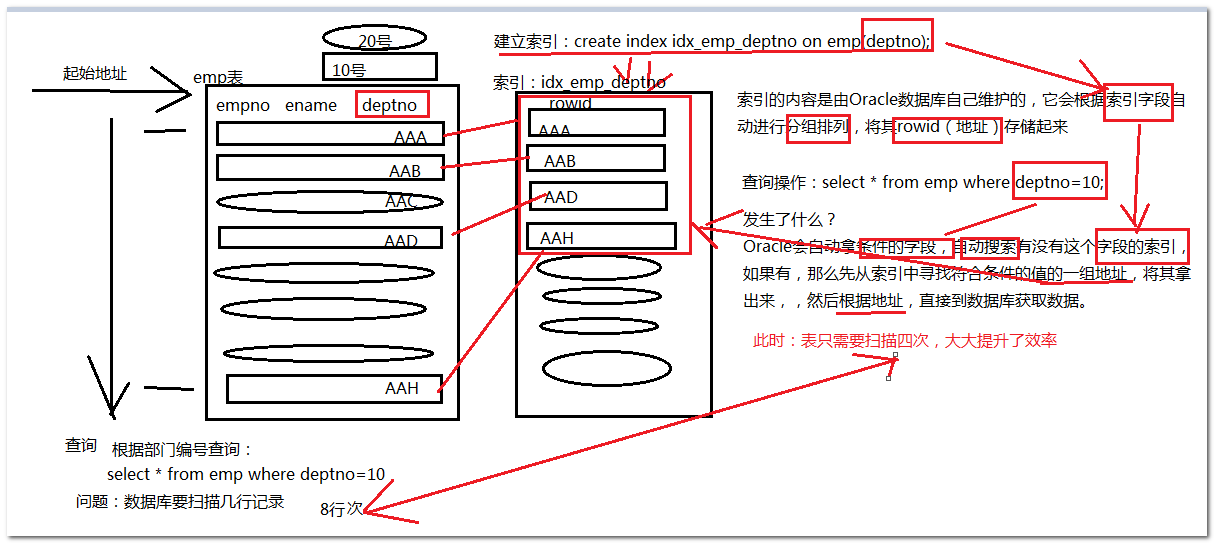
特性：

* 一种独立于表的模式（数据库）对象, 可以存储在与表不同的磁盘或表空间中。
* 索引被删除或损坏, 不会对表（数据）产生影响, 其影响的只是查询的速度。
* 索引一旦建立, Oracle 管理系统会对其进行自动维护, 而且由 Oracle 管理系统决定何时使用索引. 用户不用在查询语句中指定使用哪个索引。
* 在删除一个表时, 所有基于该表的索引会自动被删除。
* 如果建立索引的时候,没有指定表空间,那么默认索引会存储在哪个表空间.会存储在所属用户默认的表空间.

作用：

* 通过指针（地址）加速Oracle 服务器的查询速度。
* 提升服务器的i/o性能（减少了查询的次数）；

## 索引的工作原理



## 操作索引

索引的常见操作可以创建、删除。

### 创建索引

索引有两种创建方式：

* 自动创建: 在定义 PRIMARY KEY 或 UNIQUE 约束后系统自动在相应的列上创建唯一性索引。
* 手动创建: 用户可以在其它列上创建非唯一的索引，以加速查询。

唯一索引的数据不重复，一般用于主键或唯一约束上创建。

其他普通的列的数据，一般可以重复，那么就不要创建唯一索引，创建一个非唯一、普通索引。

1. 自动创建

【示例】主键自动创建索引

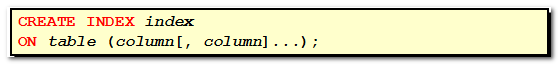
|  |
| --- |
| --在emp表的empno上增加主键PK\_EMP |

思考：

创建主键的同时自动创建索引的好处(数据库的机制)：是为了提高查询速度，一般业务中通过主键查询的比较频繁。

1. 手动创建

语法：



提示：可以在一个列或多个列上同时创建索引。

提示：如果在多个列上创建索引，也称之为联合索引。（类似联合主键的概念）

联合索引where条件中一般有索引的所有字段

【示例】手动创建索引

|  |
| --- |
| --在指定的表空间上创建索引  --使用工具：    --生成的sql：  *-- Create/Recreate indexes*  create index idx\_emp\_ename on EMP (ename);  --缺点：没有指定表空间，生产环境下一般要将索引单独指定表空间。  create index idx\_emp\_ename on EMP (ename) tablespace USERS;  --如果在生产环境下，创建表空间的脚本最好不要指定表空间的具体参数，例如将下图的深蓝色的地方去掉。 |

### 删除索引

语法：



【示例】删除上例建立的索引idx\_emp\_ename

|  |
| --- |
| Drop index idx\_emp\_ename; |

## 执行计划Explain Plan

如何来查看索引是否生效以及预估索引性能效果呢？

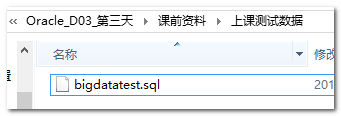
采用执行计划。这里我们使用工具的执行计划功能。（sqlplus的执行计划功能可以课后自己了解，此处不做赘述）。

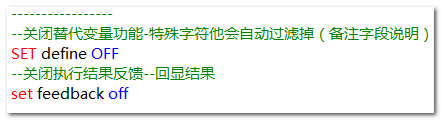
打开执行计划窗口有两种方式：

* 直接打开执行计划的空窗口，然后输入需要执行的SQL查询语句。
* 在SQL查询窗口中的语句上，按F5，可以自动打开执行计划窗口，并且会将选中的SQL语句自动填入执行计划窗口中自动执行。

下面我们演示一下如何查看索引的使用情况和效率

【准备数据】--补充知识：脚本的执行

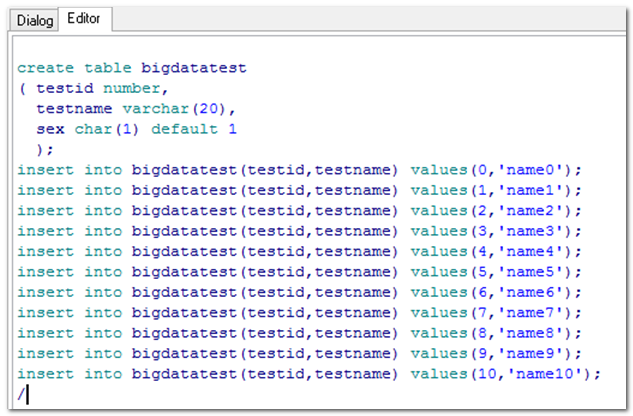




将准备好的脚本在数据库执行。执行脚本有两种方式：

1. 使用command窗口的editor子窗口执行。

将脚本直接粘贴到editor窗口，然后点击执行按钮或按F8：



注意：

命令行中执行的脚本在最后面要加上一个符号 /，标识脚本终止，可以执行了。

1. 在命令行使用@方式执行。

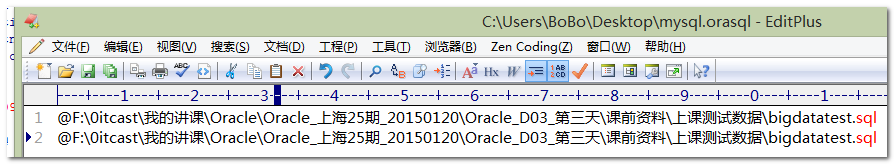
在命令行下执行：@sql脚本路径，如：

@F:\0baby\我的备课\Oracle\00上课资料\Oracle\_D03\_第三天\课前资料\课件\bigdatatest.sql



第二种方法适合大量脚本的批量执行。推荐

如果有更多的脚本文件需要同时执行，可以再建立一个文本文件，将这些脚本都引入进去，引入的方式是@子脚本



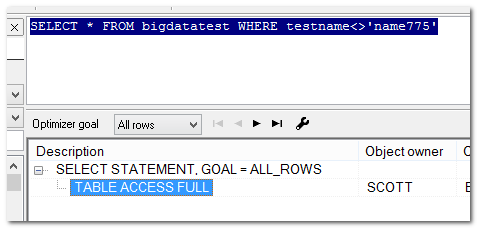
【示例】演示索引的效率查看

|  |
| --- |
| --在testname字段上创建索引    --查询语句：  --创建索引前后的结果对比： |

## 强制索引—了解

在查询字段上增加索引后，查询操作时索引一定会生效么？

将上面的语句修改一下，执行后发现索引没有生效：



效果：索引失效！

解决方案是：使用强制索引。

强制索引的语法：

/\*+INDEX(表名,索引名字)\*/

【示例】

|  |
| --- |
| SELECT */\*+index(bigdatatest,idx\_bdt\_testname)\*/* \* FROM bigdatatest WHERE testname <> 'name775';    强制索引对于超大的数据量的表来说是有一定的作用的，虽然它是全索引扫描，但扫描索引比扫描表速度还是快。而且场景：不等于的条件的值 占比比较小。（大大提升效率） |

强制索引的使用注意：

1. 尽量少用强制索引，如何避免使用，条件上尽量不要用<>,like ‘%aa%’
2. 如果要用强制索引，在非常大的数据量的情况下使用。(特殊场景使用)

## 索引的创建场景

索引不是万能！

* 以下情况可以创建索引:
  + 列中数据值分布范围很广
  + 列经常在 WHERE 子句或连接条件中出现
  + 表经常被访问而且数据量很大 ，访问的数据大概占数据总量的2%到4%
* 下列情况不要创建索引:
  + 表很小
  + 列不经常作为连接条件或出现在WHERE子句中
  + 查询的数据大于2%到4%
  + 表经常频繁更新（看需求，如果表经常不断的再更新，Oracle会频繁的重新改动索引，反而降低了数据库性能。但如系统日志历史表，就必须增加索引，效率超高）

面试题：

1. 索引的作用是什么？

主要是提高查询效率,减少磁盘的读写，从而提高数据库性能。

2. 创建索引一定能提高查询速度么？

未必！得看你创建的索引的合理性和语句的编写

3. 索引创建的越多越好么？

不是！索引也是需要占用存储空间的，过多的索引不但不会加速查询速度，反而还会降低效率。

友情提示：如果数据非常大的情况下，增加索引，一定要在业务低峰期操作，因为会花费一些时间。（排它锁）

# 数据字典DICTIONARY

## 概念

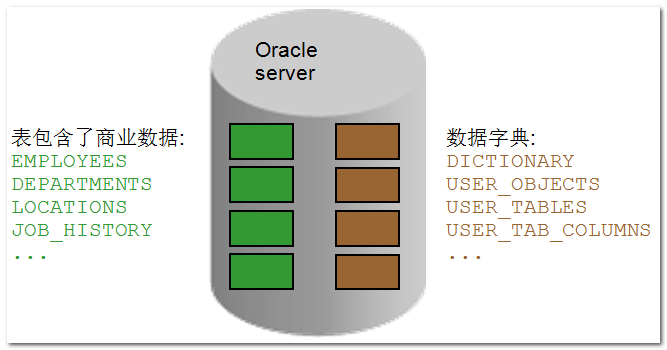
为什么要有数据字典？

数据库是数据的集合，数据库维护和管理这用户的数据，那么这些用户数据表都存在哪里，用户的信息是怎样的，存储这些用户的数据的路径在哪里，这些信息不属于用户的信息，却是数据库维护和管理用户数据的核心，这些信息就是数据库的数据字典来维护的，数据库的数据字典就汇集了这些数据库运行所需要的基础信息。

什么是数据字典？

Oracle的数据字典是Oracle数据库安装之后，自动创建的一系列表。

数据字典表和用户创建的表没有什么区别，不过数据字典表里的数据是Oracle系统存放的系统数据，而普通表存放的是用户的业务数据而已。



对于数据字典表，里面的数据是有数据库系统自身来维护的。所以这里虽然和普通表一样可以用DML语句来修改数据内容，但是大家最好还是不要自己来做了，因为这些表都是作用于数据库内部的。所以这里我们切记记住不要去修改这些表里的内容。，所以，数据字典主要用来查询的。

## 数据字典的命名规则

数据字典表的用户都是sys，存在在system这个表空间里，表名都用"$"结尾，为了便于用户对数据字典表的查询（这样的名字是不利于我们记忆的），所以Oracle对这些数据字典都分别建立了用户视图，不仅有更容易接受的名字，还隐藏了数据字典表表之间的关系，让我们字节通过视图来进行查询，简单而形象，Oracle针对这些对象的范围，分别把视图命名为DBA\_XXXX, ALL\_XXXX和USER\_XXXX，所以我们说的数据字典一般是指数据字典视图。

|  |  |
| --- | --- |
| 视图名称 | 作用 |
| user\_对象视图 | 当前用户schema（方案-和用户同名）下的创建对象； |
| all\_对象视图 | 当前用户有权限访问到的所有对象的信息； |
| dba\_对象视图 | 管理员视图，包括了所有数据库对象的信息； |
| v$\_ | 性能视图，查询数据库性能使用。 |

数据字典视图非常多，我们无法一一记住，但是有个视图，我们必须知道，那就是dictionary视图，该视图里记录了所有的数据字典视图的名称。所以当我们需要查找某个数据字典而又不知道这个信息在哪个视图里的时候，就可以在dictionary视图里找。该视图还有个同义词dict。

【示例】

需求1：查询所有数据字典的名称和描述

需求2：我想查看当前用户下有哪些视图，但我不知道查询当前用户视图的数据字典的名称

|  |
| --- |
| *--需求1：查询所有数据字典的名称和描述*  SELECT \* FROM DICTIONARY;*--视图*  SELECT \* FROM dict;*--同义词*  *--需求2：我想查看当前用户下有哪些视图对象，但我不知道查询当前用户视图的数据字典的名称*  SELECT \* FROM DICTIONARY WHERE table\_name LIKE UPPER('user\_%view%');  SELECT \* FROM User\_Views;*--当前用户下创建的所有视图* |

数据字典中的表名默认是大写的！！！

## 常用的数据字典

|  |  |
| --- | --- |
| 数据字典名称 | 作用 |
| USER\_OBJECTS | 当前用户所创建数据库对象。 |
| ALL\_OBJECTS | 当前用户能够访问的数据库对象 |
| USER\_TABLES | 当前用户创建的表对象的信息 |
| USER\_TAB\_COLUMNS | 当前用户下创建的列信息。 |
| USER\_CONSTRAINTS | 当前用户表上的约束对象 |
| USER\_CONS\_COLUMNS | 当前用户下创建的列约束对象 |
| USER\_VIEWS | 当前用户下创建的视图 |
| USER-SEQUENCES | 当前用户下创建的序列 |
| USER-SYNONYMS | 当前用户下创建的同义词 |
|  |  |

【示例】

需求：由于业务需要，查询当前用户下有没有emp这张表（如果没有就创建，有的话就直接插入数据）

|  |
| --- |
| --需求：由于业务需要，查询当前用户下有没有emp这张表（如果没有就创建，有的话就直接插入数据）  SELECT \* FROM user\_tables WHERE table\_name =UPPER('emp');  *--我想看看emp表中有几列，将列名都打印出来*  SELECT \* FROM USER\_TAB\_COLUMNS WHERE table\_name =UPPER('emp'); |

[扩展]：利用数据字典自己编写一个类似plsql管理Oracle的网页版工具。

# PLSQL编程

## 概念和目的

什么是PL/SQL?

* PL/SQL（Procedure Language/SQL）
* PLSQL是Oracle对sql语言的过程化扩展
* 指在SQL命令语言中增加了过程处理语句（如分支、循环等），使SQL语言具有过程处理能力。（具有编程的能力）

为什么要学习plsql？

1. 将sql逻辑写在db层，效率更高----数据库处理数据更专业，还不需要网络数据交换。
2. 为存储过程、函数等打下基础，前提是学会plsql

## Hello World

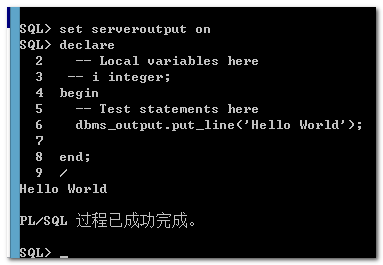
通过程序在控制台打印一句话

一般我们调试或者练习plsql程序可以用test窗口（专门用来调试或练习的）

--建议在test窗口进行练习

|  |
| --- |
| *--过程化的语言，类似与basic*  declare  begin  *--打印一句话*  *--下面这句话相当于java：System.out.println("hello world");*  *--dbms\_output:Oracle的内置包，相当于java类*  *--put\_line：方法，相当于println*  dbms\_output.put\_line('hello world');  end;    --点击运行或者F8 |

如果想要在命令窗口运行，需要打开输出选项：set serveroutput on



注意：oracle的控制台信息，默认不会显示到客户端，需要设置set serveroutput on，目的是打开控制台信息的输出。

先分析一下：

|  |
| --- |
| --面向过程的语言  --declare --声明部分：没有变量，则declare可以省略  --你不需要变量声明，则不需要写任何东西  BEGIN--程序体的开始：编写语句逻辑  --在控制台输出一句话：dbms\_output相当于system.out类,内置程序包，put\_line:相当于println()方法  dbms\_output.put\_line('Hello World');  --dbms\_output.put('Hello World');  end;--程序体的结束 |

概念：程序包：dbms\_output相当于java中的类(system.out),它是oracle自带的,内置.

调用程序包：dbms\_output.put\_line(‘Hello World!’)相当于java的方法

## 程序结构

PL/SQL可以分为三个部分：声明部分、可执行部分、异常处理部分。

|  |
| --- |
| [delare]  声明部分（变量、游标、例外）  begin  逻辑执行部分（DML语句、赋值、循环、条件等）  [exception]  异常处理部分（when 预定义异常错误 then）  end;  / |

最简单的PL/SQL：

|  |
| --- |
| Begin  Null;  End;  / |

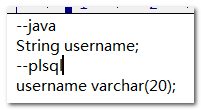
注意：在SQLPLUS中，PLSQL执行时，要在最后加上一个 **“/”**

## 变量

### 语法说明

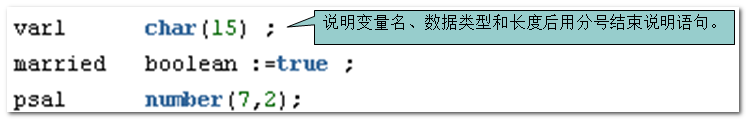
声明部分可以定义变量，定义变量的语法：

|  |
| --- |
| 变量名 [CONSTANT] 数据类型; |

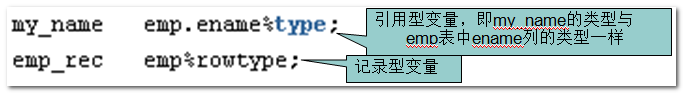


常见的几种变量类型的定义方法，分两大类：

* 普通数据类型（char, varchar2, date, number, boolean, long）：



* 特殊变量类型（引用型变量、记录型变量）：



### 普通变量赋值

在ORACLE中有两种赋值方式：

1，直接赋值语句 :=

2, 使用select …into … 赋值：（语法;select 值 into 变量）

【示例】

打印几个变量的值，几个变量的值分别采用两种不同的赋值方法：

|  |
| --- |
| *--打印两个变量的值，两个变量的值分别采用两种不同的赋值方法：*  DECLARE*--声明变量*  *--姓名*  v\_name VARCHAR(20) :='Bo';*--声明的时候直接赋值*  *--薪资*  v\_sal NUMBER;  *--工作地点*  v\_local VARCHAR(200);  BEGIN *--开始程序逻辑*  *--程序运行时赋值*  *--方法一：--直接赋值*  v\_sal :=9999;  *--方法二：语句赋值*  SELECT '上海' INTO v\_local FROM dual;  *--输出打印*  dbms\_output.put\_line('姓名：'||v\_name||',薪资：'||v\_sal||'，工作地点：'||v\_local);  END;*--程序结束* |

plsql中有两种不同的赋值方法：

一种是： 直接用:=来赋值

另一种：select 值 into 变量 from 表名。

### 引用型变量

引用变量：引用表中字段的类型 （推荐使用引用类型）

%type 例： v\_ename emp.ename%type;

使用emp表的字段ename的数据类型作为v\_ename的数据类型.

【示例】

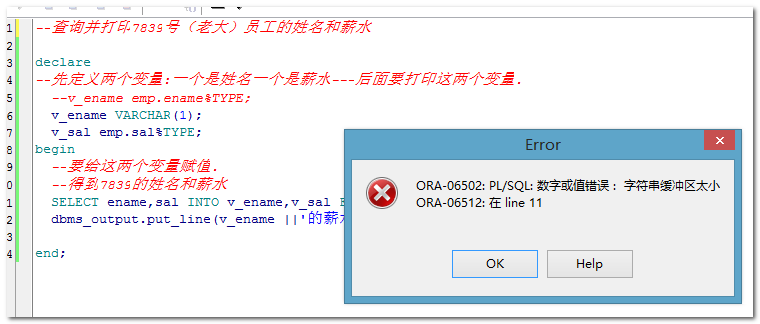
查询并打印7839号（老大）员工的姓名和薪水

|  |
| --- |
| *--查询并打印7839号（老大）员工的姓名和薪水*  DECLARE  *--定义变量*  *--姓名*  v\_ename emp.ename%TYPE;*--姓名使用的emp表中的ename的字段的数据类型*  *--薪水*  v\_sal emp.sal%TYPE;*--你不需要关心具体什么数据类型了*  BEGIN  *--赋值*  *--注意：into前后字段名和变量名必须对应（不管是数据类型，还是个数，顺序）*  *--必须：查询的结果必须只有一个值，不能有多行记录*  SELECT ename,sal INTO v\_ename,v\_sal FROM emp WHERE empno=7839;  *--打印*  dbms\_output.put\_line('7839号员工的姓名是：'||v\_ename||'，薪资'||v\_sal);  END; |

引用类型的好处：

1. 使用普通变量定义方式，需要知道表中列的类型，而使用引用类型，不需要考虑列的类型

普通变量值过小报错:



1. 使用引用类型，当列中的数据类型发生改变，不需要修改变量的类型。而使用普通方式，当列的类型改变时，需要修改变量的类型

使用%TYPE是非常好的编程风格，因为它使得PL/SQL更加灵活，更加适应于对数据库定义的更新。

### 记录型变量

**记录型变量，代表一行，**可以理解为数组，里面元素是每一字段值。

%rowtype 引用一条(行)记录的类型 例：v\_emp emp%rowtype;

含义：v\_emp 变量代表emp表中的一行数据的类型，它可以存储emp表中的任意一行数据。

记录型变量分量的引用方式：



【示例】

查询并打印7839号（老大）员工的姓名和薪水

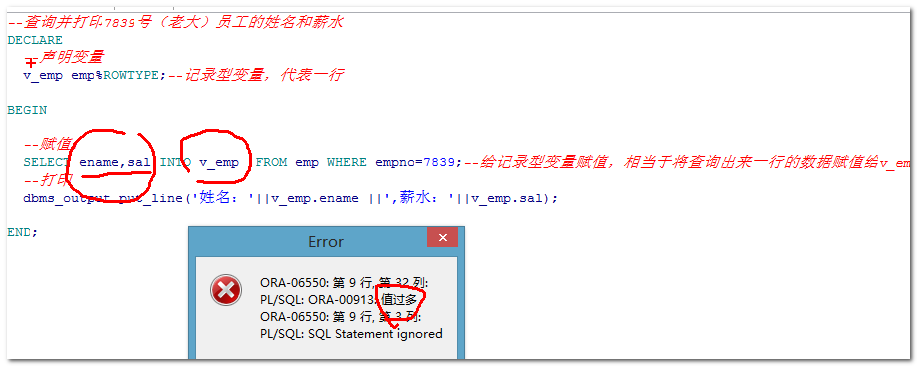
|  |
| --- |
| *--查询并打印7839号（老大）员工的姓名和薪水*  DECLARE  *--记录型变量*  v\_emp emp%ROWTYPE;*--该变量可以存储emp表中的一行记录*  BEGIN  *--赋值*  *--默认情况下，必须是全字段赋值*  SELECT \* INTO v\_emp FROM emp WHERE empno=7839;  *--打印*  dbms\_output.put\_line('7839号员工的姓名是：'||v\_emp.ename||'，薪资'||v\_emp.sal);  END; |

场合：

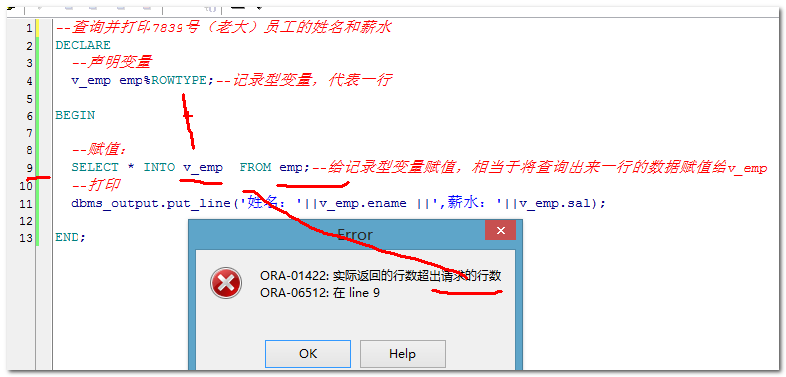
如果有一个表，有100个字段，那么你程序如果要使用这100字段话，如果你使用引用型变量一个个声明，会特别麻烦，那么你可以考虑记录型变量

错误的使用：

1. 记录型变量只能存储一个完整的行数据



2．返回的行太多了，记录型变量也接收不了



## 流程控制

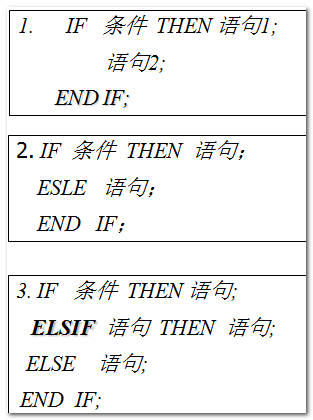
可以执行就是对PL/SQL进行程序控制

程序控制：

1. 顺序结构
2. 条件结构
3. 循环结构

### 条件分支IF

语法：



提示：语法和java作用差不多。

java三种if语法：

if(){

}

if(){

}else{  
}

if(){

}elseif(){  
}else{  
}

注意单词 。

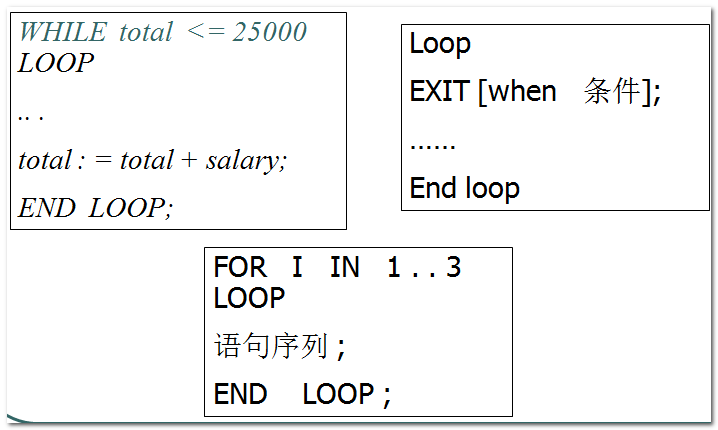
【示例】

判断emp表中记录是否超过20条，，10-20之间，10以下打印一句

|  |
| --- |
| *--判断emp表中记录是否超过20条，，10-20之间，10以下打印一句*  DECLARE  *--用来存储数量*  v\_count NUMBER;  BEGIN  *--查询数量赋值*  SELECT COUNT(1) INTO v\_count FROM emp ;  *--判断*  IF v\_count>20 THEN  dbms\_output.put\_line('记录数超过20条:'||v\_count);  ELSIF v\_count BETWEEN 10 AND 20 THEN  dbms\_output.put\_line('记录数在10到20条之间:'||v\_count);  ELSE  dbms\_output.put\_line('记录数不足10条:'||v\_count);  END IF;  END; |

### 循环

语法：



在ORACLE中有三种循环：

Loop 循环 EXIT WHEN...条件 end loop;

While()…loop 条件判断循环

For 变量 in 起始..终止 Loop

这里我建议只记忆一种写法:

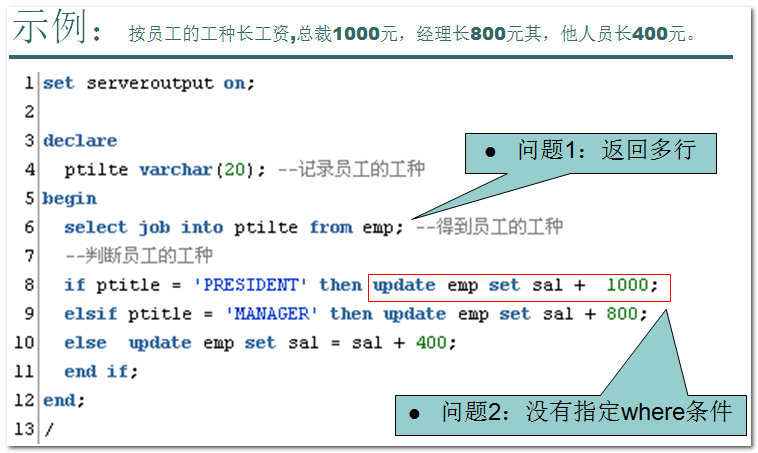
记住loop的写法

【示例】

打印数字1-10

|  |
| --- |
| *--打印数字1-10*  DECLARE  *--声明一个变量*  v\_num NUMBER :=1;  BEGIN  *--循环并打印*  LOOP  EXIT WHEN v\_num>10; *--退出循环条件*  dbms\_output.put\_line(v\_num);  *--递增*  *--v\_num++;--不支持*  v\_num :=v\_num+1;  END LOOP;  END; |

## 错误的编程语句举例



纠正：(课后练习)

# 游标（Cursor）

## 什么是游标

|  |
| --- |
| 游标（Cursor），也称之为光标,从字面意思理解就是游动的光标。  游标是映射在结果集中一行数据上的位置实体。  游标是从表中检索出结果集，并从中每次指向一条记录进行交互的机制。 |

游标从概念上讲基于数据库的表返回结果集，也可以理解为游标就是个结果集，但该结果集是带向前移动的指针的，每次只指向一行数据。



游标的主要作用：用于临时存储一个查询返回的多行数据（结果集），通过遍历游标，可以逐行访问处理该结果集的数据。

（显示）游标的使用方式：声明--->打开--->读取--->关闭

## 语法

游标声明：

|  |
| --- |
| CURSOR 游标名 [ (参数名 数据类型[,参数名 数据类型]...)]  IS SELECT 语句;  【示例】  无参游标：  cursor c\_emp is select ename from emp;  有参游标：  cursor c\_emp(v\_deptno emp.deptno%TYPE) is select ename from emp where deptno=v\_deptno; |

游标的打开：

|  |
| --- |
| Open 游标名(参数列表)  【示例】  open c\_emp;-- 打开游标执行查询 |

游标的取值：

|  |
| --- |
| fetch 游标名 into 变量列表|记录型变量  【示例】  fetch c\_emp into v\_ename;--取一行游标的值到变量中，注意：v\_ename必须与emp表中的ename列类型一致。(v\_ename emp.ename%type;) |

游标的关闭：

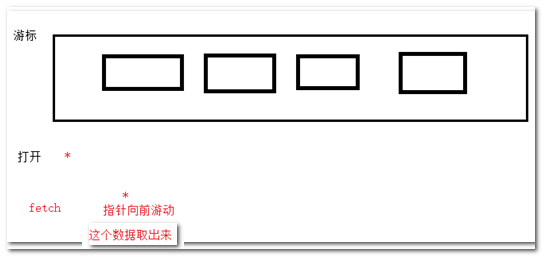
|  |
| --- |
| close 游标名  【示例】  close c\_emp;--关闭游标释放资源 |

解释游标获取数据的基本原理：

游标刚open的时候，指针结果集的第一条记录之前。

游标与结果集的区别是什么?游标是有位置的。

fetch会向前游动，并获取游标的位置的内容。



注意：游动过的就不能回来了，循环一次就到头。

## 游标的属性

|  |  |  |
| --- | --- | --- |
| 游标的属性 | 返回值类型 | 说明 |
| %ROWCOUNT | 整型 | 获得FETCH语句返回的数据行数 |
| %FOUND | 布尔型 | 最近的FETCH语句返回一行数据则为真，否则为假 |
| %NOTFOUND | 布尔型 | 与%FOUND属性返回值相反，代表游标结束 |
| %ISOPEN | 布尔型 | 游标已经打开时值为真，否则为假 |

## 创建和使用

【示例】

使用游标查询emp表中所有员工的姓名和工资，并将其依次打印出来。

【引用型变量获取游标的值】：

|  |
| --- |
| *--使用游标查询emp表中所有员工的姓名和工资，并将其依次打印出来。*  DECLARE  *--声明一个游标*  CURSOR C\_EMP IS  SELECT ENAME, SAL FROM EMP;  *--引用型变量*  V\_ENAME EMP.ENAME%TYPE; *--姓名*  V\_SAL EMP.SAL%TYPE; *--工资*  BEGIN  *--打开游标，执行查询*  OPEN C\_EMP;  *--使用游标，循环取值*  LOOP  *--获取游标的值放入变量的时候，必须要into前后要对应（数量和类型）*  FETCH C\_EMP INTO V\_ENAME, V\_SAL;  EXIT WHEN C\_EMP%NOTFOUND;  *--输出打印*  DBMS\_OUTPUT.PUT\_LINE('员工的姓名：' || V\_ENAME || '，员工的工资' || V\_SAL);  END LOOP;  CLOSE c\_emp ;*--关闭游标，释放资源*  END; |

【使用记录型变量存值】：

|  |
| --- |
| *--使用游标查询emp表中所有员工的姓名和工资，并将其依次打印出来。*  DECLARE  *--声明一个游标*  CURSOR C\_EMP IS SELECT \* FROM EMP;  *--记录型变量*  v\_emp emp%ROWTYPE;  BEGIN  *--打开游标，执行查询*  OPEN C\_EMP;  *--使用游标，循环取值*  LOOP  *--获取游标的值放入变量的时候，必须要into前后要对应（数量和类型）*  FETCH C\_EMP INTO v\_emp;  EXIT WHEN C\_EMP%NOTFOUND;  *--输出打印*  DBMS\_OUTPUT.PUT\_LINE('员工的姓名：' || v\_emp.ename || '，员工的工资' || v\_emp.sal);  END LOOP;  CLOSE c\_emp ;*--关闭游标，释放资源*  END; |

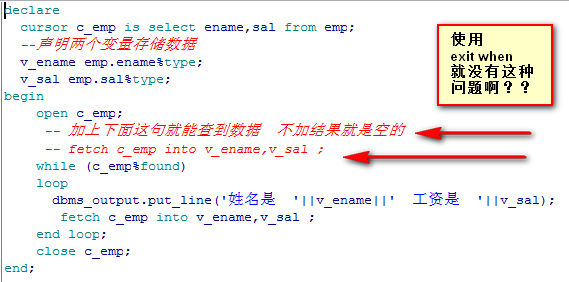
## 带参数的游标

【示例】

使用游标查询并打印某部门的员工的姓名和薪资，部门编号为运行时手动输入。

|  |
| --- |
| *-- Created on 2015/5/5 by BOBO*  *---查询10号部门的员工的姓名和薪资*  declare  *--定义游标--带参数的游标:需要定一个形式参数*  CURSOR c\_emp(v\_deptno emp.deptno%TYPE) IS SELECT ename,sal FROM emp WHERE deptno=v\_deptno ;    *--声明变量*  v\_ename emp.ename%TYPE;  v\_sal emp.sal%TYPE;  BEGIN  *--用*  *--打开游标*  OPEN c\_emp(10);  *--循环fetch*  LOOP  *--取出数据*  FETCH c\_emp INTO v\_ename,v\_sal;  *--退出条件*  EXIT WHEN c\_emp%NOTFOUND;  *--打印--写任何的逻辑*  dbms\_output.put\_line('姓名：'||v\_ename||'，薪资：'||v\_sal);  END LOOP;  *--关闭*  CLOSE c\_emp;  end;  *--使用游标查询并打印某部门的员工的姓名和薪资，部门编号为运行时手动输入。*  DECLARE  *--声明一个带参数的游标*  CURSOR C\_EMP(v\_deptno emp.deptno%TYPE) IS SELECT \* FROM EMP WHERE deptno=v\_deptno;  *--记录型变量*  v\_emp emp%ROWTYPE;  BEGIN  *--打开游标，执行查询*  *--打开游标的时候需要传入参数*  OPEN C\_EMP(20);  *--使用游标，循环取值*  LOOP  *--获取游标的值放入变量的时候，必须要into前后要对应（数量和类型）*  FETCH C\_EMP INTO v\_emp;  EXIT WHEN C\_EMP%NOTFOUND;  *--输出打印*  DBMS\_OUTPUT.PUT\_LINE('员工的姓名：' || v\_emp.ename || '，员工的工资' || v\_emp.sal);  END LOOP;    CLOSE c\_emp ;*--关闭游标，释放资源*  END; |

【瑞瑞】



* Found是游标有数据的判断。如果游标刚打开，值false
* Notfound是游标结束的判断，如果游标刚打开，值false，只判断游标是否结束！

# 存储过程

## 概念作用

存储过程：就是一块PLSQL语句包装起来，起个名称

语法上:相当于plsql语句戴个帽子。

相对而言:单纯plsql可以认为是匿名程序。

提示：

* plsql是存储过程的基础。
* java是不能直接调用plsql的，但可以通过存储过程这些对象来调用。

## 语法



As和is是通用的。

根据参数的类型，我们将其分为3类讲解：

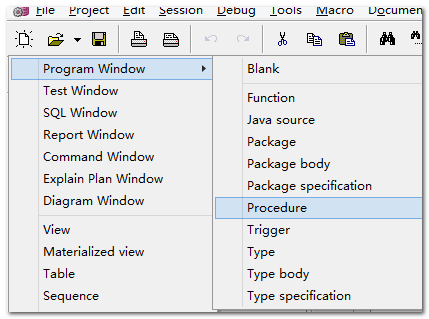
* 不带参数的
* 带输入参数的
* 带输入输出参数的。

## 无参存储

### 创建存储

最简单，就是包装了一个代码块

建议用这个窗口:



【示例】

|  |
| --- |
| create or replace procedure p\_hello--没有参数的情况下，不要加()  IS  BEGIN--plsql程序  dbms\_output.put\_line('hello world');  end p\_hello; |

查询是否创建：

在工具procedures这里



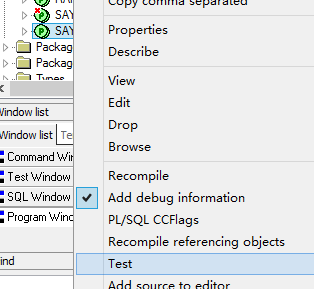
关于写存储的3个窗口的选择：

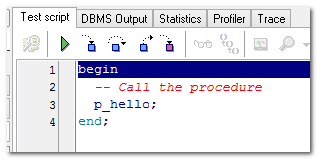


编译发布的时候使用command、测试使用test

### 调试存储

测试一下：

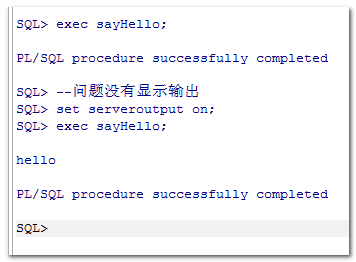




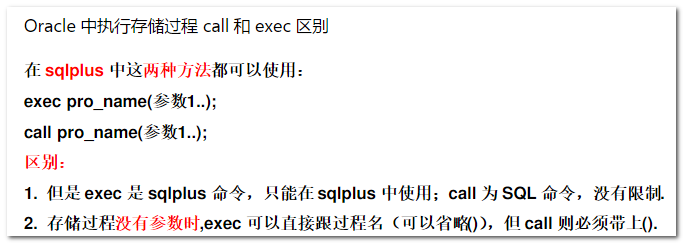
### 调用方法

如何调用执行，两种方法：

* 一种是是用exec（execute）命令来调用—用来测试存储
* 一种是用其他的程序（plsql和java）来调用
* 命令调用的方式:



【扩展】



* 程序调用

BEGIN

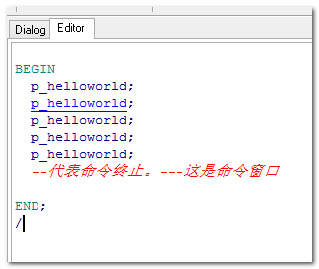
sayhelloworld;

sayhelloworld;

sayhelloworld;

END;

/



注意：

第一个问题：is和as是可以互用的，用哪个都没关系的  
第二个问题：存储过程中没有declare关键字，declare用在语句块中

存储可以带参数可以不带参数？其实都有应用.

不带参数的存储一般用来处理内部数据的。不需要输入参数也不需要结果的，是可以使用。

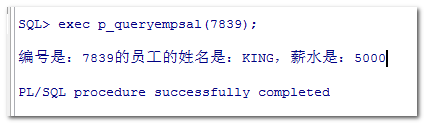
## 带输入参数in

【示例】

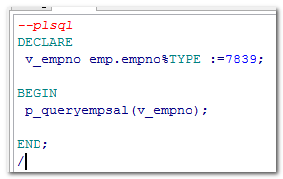
查询并打印某个员工（如7839号员工）的姓名和薪水--存储过程：要求，调用的时候传入员工编号，自动控制台打印。

|  |
| --- |
| *--查询并打印某个员工（如7839号员工）的姓名和薪水--存储过程：要求，调用的时候传入员工编号，自动控制台打印。*  create or replace procedure p\_queryempsal(i\_empno IN emp.empno%TYPE)*--i\_empno输入参数的名字，IN代表是输入值的参数，*  IS  *--声明变量*  v\_ename emp.ename%TYPE;  v\_sal emp.sal%TYPE;  BEGIN  *--赋值*  SELECT ename ,sal INTO v\_ename,v\_sal FROM emp WHERE empno= i\_empno;  *--打印*  dbms\_output.put\_line('姓名：'||v\_ename||'，薪水：'||v\_sal);  end p\_queryempsal; |

命令调用：



程序调用：



## 带输入in和输出参数out—主要是其他程序用的。

【示例】

输入员工号查询某个员工（7839号员工）信息，要求，将薪水作为返回值输出，给调用的程序使用。

|  |
| --- |
| *----输入员工号查询某个员工（7839号（老大）员工）信息，要求，将薪水作为返回值输出，给调用的程序使用。*  CREATE OR REPLACE PROCEDURE p\_queryempsal\_out( i\_empno IN emp.empno%TYPE,o\_sal OUT emp.sal%TYPE)  AS  BEGIN  *--赋值:将薪水的值赋给输出的参数o\_sal*  SELECT sal INTO o\_sal FROM emp WHERE empno=i\_empno;  END; |

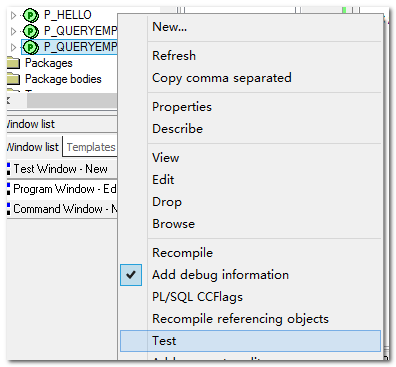
调用（使用plsql程序调用）：

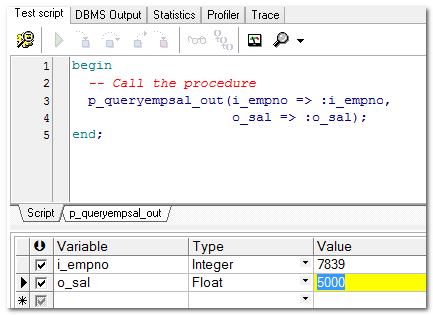
|  |
| --- |
| DECLARE  *--输入参数值*  v\_empno emp.empno%TYPE:=7839;  *--声明一个变量来接收输出参数*  v\_sal emp.sal%TYPE;  BEGIN  p\_queryempsal\_out(v\_empno,v\_sal);*--第二个参数是输出的参数，必须有变量来接收！！*  *--当上面的语句执行之后，v\_sal就有值了。*    dbms\_output.put\_line('员工编号为：'||v\_empno||'的薪资为：'||v\_sal);  END;  / |

注意:调用的时候,参数要与定义的参数的顺序和类型一致.

【扩展】

如何直接测试存储（相当于debug）测试：

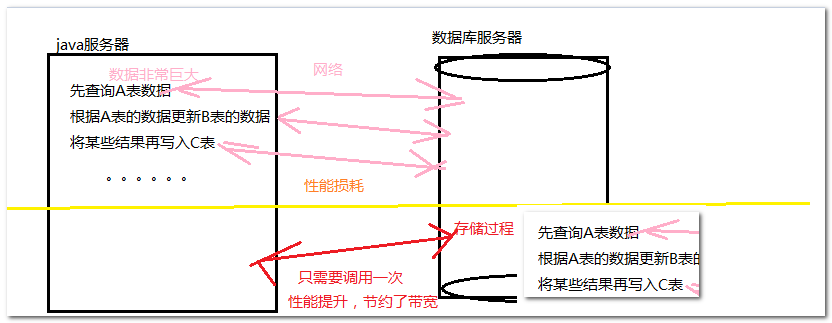




小结：

存储过程作用：主要用来执行一段程序。

1. 在开发程序中，为了一个特定的业务功能，会向数据库进行多次连接关闭(连接和关闭是很耗费资源)。这种就需要对数据库进行多次I/O读写，性能比较低。如果把这些业务放到PLSQL中，在应用程序中只需要调用PLSQL就可以做到连接关闭一次数据库就可以实现我们的业务,可以大大提高效率.



1. ORACLE官方给的建议：能够让数据库操作的不要放在程序中。在数据库中实现基本上不会出现错误，在程序中操作可以会存在错误.(如果在数据库中操作数据,可以有一定的日志恢复等功能.)

三种存储基本应用场景：

* 无参参数：只要用来做数据处理的。存储内部写一些处理数据的逻辑。
* 带输入参数：数据处理时，可以针对输入参数的值来进行判断处理。
* 带输入输出参数：一般用来传入一个参数值，我想经过数据库复杂逻辑处理后，得到我想要的值然后输出给我。

## 存储函数-了解

存储函数创建语法：

|  |
| --- |
| **CREATE [OR REPLACE] FUNCTION 函数名(参数列表)**  **RETURN 函数值类型**  **AS**  **PLSQL子程序体；** |

存储函数的编写示例：

|  |
| --- |
| /\*  查询某职工的总收入。  \*/  create or replace function queryEmpSalary(i\_empid in number)  RETURN NUMBER  as  pSal number; --定义变量保存员工的工资  pComm number; --定义变量保存员工的奖金  begin  select sal,comm into pSal, pcomm from emp where empno = i\_empid;  return psal\*12+ nvl(pcomm,0);  end;  / |

存储过程的调用：

|  |
| --- |
| declare  v\_sal number;  begin  v\_sal:=queryEmpSalary(7934);  dbms\_output.put\_line('salary is:' || v\_sal);  end;  / |
| begin  dbms\_output.put\_line('salary is:' || queryEmpSalary(7934));  end; |

存储过程和存储函数的区别：

|  |
| --- |
| * 一般来讲，过程和函数的区别在于函数可以有一个返回值；而过程没有返回值。 * 但过程和函数都可以通过out指定一个或多个输出参数。我们可以利用out参数，在过程和函数中实现返回多个值。 |

如何选择存储过程和存储函数？

原则上，如果只有一个返回值，用存储函数，否则，就用存储过程。

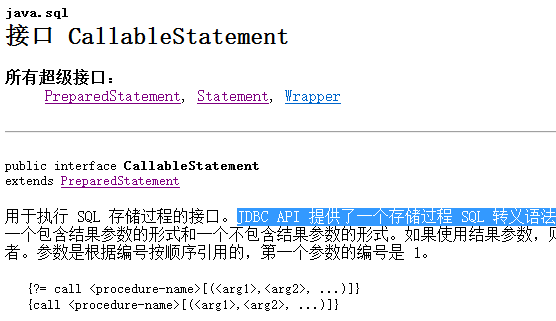
但是，一般我们会直接选择使用存储过程，原因是：

* 函数是必须有返回值，存储可以有也可以没有，存储的更灵活！
* 既然存储也可以有输出参数,可以代替存储函数。.
* Oracle的新版本中，已经不推荐使用存储函数了。

## java程序调用

需求:如果一条语句无法实现结果集的查询，(比如需要多表查询,或者需要复杂逻辑查询,),你可以选择调用存储查询出你的结果.

### 分析jdk API



调用存储的语句：转义语法



如何得到？

通过connection得到:



### 代码

准备环境：

* 导入Oracle的jar包
* 导入jdbcutil类

【示例】

通过员工号查询员工的姓名和薪资

|  |
| --- |
| //获取连接  Connection conn = JDBCUtils.*getConnection*();    //获取CallableStatement  //{call <procedure-name>[(<arg1>,<arg2>, ...)]}  String sql="{call p\_queryempsal\_out(?,?)}";//转义sql  CallableStatement call = conn.prepareCall(sql);    //设置占位符的值（区分输入和输出）  //输入参数的值  call.setInt(1, 7839);  //输出参数，需要注册  //第二个参数是java和数据库对应的类型  //java.sql.Types:标准sql99的类型  //推荐使用数据库自己的类型Oracle.jdbc.driver.OracleTypes  // call.registerOutParameter(2, Types.)  call.registerOutParameter(2, OracleTypes.*DOUBLE*);    //执行存储过程(将存储执行的动作发向数据库了，将执行的结果返回并封装回call对象中。) call.execute();//执行的时候，会自动将参数传入数据库,将输出参数返回的数据，封装会call对象中。    //获取输出参数的值  **double** sal = call.getDouble(2);  System.*out*.println("薪资是："+sal);    //释放资源  JDBCUtils.*release*(conn, call, **null**); |



步骤：

1. 得到连接conn
2. 从连接中获取CallableStatement对象，获取的时候，需要传入sql，sql的写法：



1. 设置占位符：要分清输入参数和输出参数：输入参数直接赋值，输出参数需要注册（注册的类型是根据数据库自己的类型来定的，oralce,----OracleTypes.）
2. 执行CallableStatement对象，向数据库调用存储过程了。
3. 从CallableStatement对象中得到输出参数的值。搞定！

# 触发器（trigger）

## 概念和作用

|  |
| --- |
| 数据库触发器是一个与表相关联的、存储的PL/SQL程序。每当一个特定的数据操作语句(Insert,update,delete)在指定的表上发出时，Oracle自动地执行触发器中定义的语句序列。 |

解释：

首先，它也是一段plsql程序。

然后，它是来触发与表数据操作相关的（insert，update，delete）。

然后，在进行表数据操作的时候，会自动触发执行的一段程序。

换句话说：触发器就是在执行某个操作（增删改）的时候触发一个动作(一段程序)。

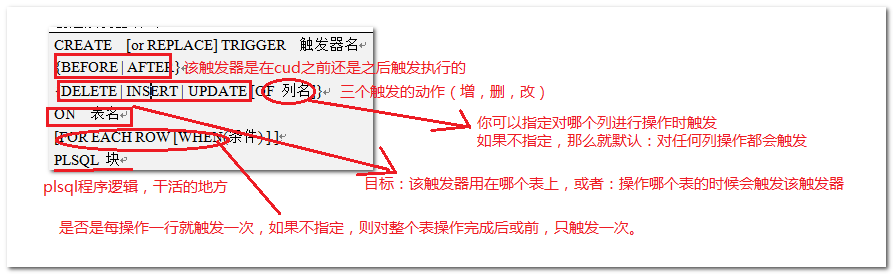
有点像struts2的拦截器，可以对cud增强。

## 语法

创建触发器语法：

|  |
| --- |
| CREATE [or REPLACE] TRIGGER 触发器名  {BEFORE | AFTER}  {DELETE | INSERT | UPDATE [OF 列名]}  ON 表名  [FOR EACH ROW [WHEN(条件) ] ]  PLSQL 块 |

解释：



## 第一个触发器

【示例 】

-每当dept表中添加了一个新部门时，打印”成功插入新部门”

|  |
| --- |
| 打开窗口：    create or replace trigger tri\_adddept  AFTER INSERT  on dept  declare  begin  dbms\_output.put\_line('插入了新部门');  end ;  --测试哈  SELECT \* FROM dept;  INSERT INTO dept VALUES(80,'baby1','上海');  SELECT \* FROM dept; |

## 触发器的类型

|  |
| --- |
| * 语句级触发器（表级触发器）   在指定的操作语句操作之前或之后执行一次，不管这条语句影响了多少行 。   * 行级触发器（FOR EACH ROW）   触发语句作用的每一条记录都被触发。在行级触发器中使用old和new伪记录变量, 识别值的状态。 |

## 语句级触发器和行级触发器的区别

【示例】目标：演示语句级触发器和行级触发器的区别

复制出来一张表depttemp，分别建立语句级和行级触发器，然后进行批量插入操作测试。

|  |
| --- |
| CREATE TABLE depttemp AS SELECT \* FROM dept WHERE 1<>1;  SELECT \* FROM depttemp; |

两个触发器编写：

|  |
| --- |
| --语句级别  create or replace trigger tri\_adddepttemp\_yuju  after insert on depttemp  declare  begin*--plsql语句*  dbms\_output.put\_line('成功插入了一个部门：语句级触发器触发了。。:');  end tri\_adddepttemp\_yuju;  --行级别：  create or replace trigger tri\_adddepttemp\_hangji  after insert on depttemp  for each row  declare  begin*--plsql语句*  dbms\_output.put\_line('成功插入了一个部门：行级触发器触发了。。:');  end tri\_adddepttemp\_hangji; |

批量插入数据测试：

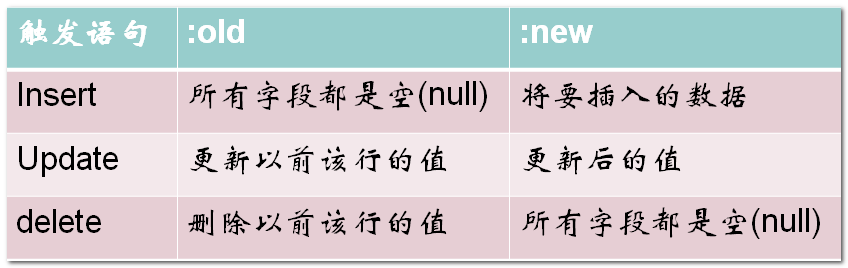
|  |
| --- |
| *--先建立两种触发器*  *--批量插入数据*  INSERT INTO depttemp SELECT \* FROM dept; |

语句级触发器和行级触发器区别：

1. 在语法上，行级触发器就多了一句话：for each row
2. 在表现上，行级触发器，在每一行的数据进行操作的时候都会触发。但语句级触发器，对表的一个完整操作才会触发一次。

简单的说：行级触发器，是对应行操作的；语句级触发器，是对应表操作的。

## 行级别触发器的伪记录变量



：new代表操作之后的数据，只出现在INSERT/UPDATE中，

：old代表操作（cud）之前的那条数据，出现在UPDATE/DELETE，

INSERT时:NEW表示新插入的行数据，UPDATE时:NEW表示要替换的新数据，:OLD表示要被更改的原来数据，DELETE时:OLD表示要被删除的数据。

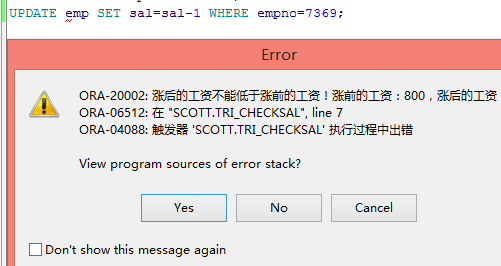
【示例】

涨工资：涨后的工资不能少于涨前的工资

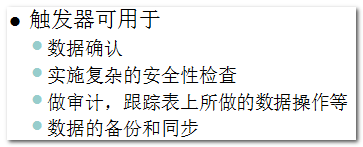
分析：行级触发器，数据确认示例--行级触发器

|  |
| --- |
| *--涨工资：涨后的工资不能少于涨前的工资*  create or replace trigger tri\_checkempsal  BEFORE UPDATE ON emp*--更新之前拦截触发*  for each row*--行级触发器*  declare  BEGIN  *--如果涨后小于涨前，则，终止更新操作*  IF :new.Sal<:old.Sal THEN  *--终止程序继续运行，也就终止了更新操作了。*  raise\_application\_error(-20001,'涨后的工资不能少于涨前的工资！！涨前的工资：'||:old.Sal||'，涨后的工资：'||:new.sal);  *--相当于抛出异常（throw）,(使用了oracle内置的一个函数来抛出异常)*  END IF;  end tri\_checkempsal; |

测试：



## 触发器的应用场景



【示例】

数据的备份：

业务的原理：在更新或者删除数据的时候，将旧的数据备份出来到另外一张表中。

|  |
| --- |
| 建立一张备份表：      目标：在更新dept的时候，进行触发该动作。（备份数据）  --自动备份数据  CREATE OR REPLACE TRIGGER tri\_autobakdept  AFTER--语句执行之后备份（已经更新了再备份，如果不更新就不备份）  UPDATE  ON dept  FOR EACH ROW--行级触发器  BEGIN  --备份：  INSERT INTO DETPBAK VALUES(seq\_test.nextval,:OLD.DEPTNO,:old.Dname,:old.Loc,  :new.Deptno,:new.Dname,:new.Loc,SYSDATE);  END;  测试： |

但是要注意：触发器会引起锁，降低效率！使用时要慎重。如无必要，尽量不要使用触发器。

行级触发器会引发行级锁（锁行数据）

语句级触发器可能会引起表级锁（锁表）

【示例】

在插入数据的之前，自动插入主键值（值是序列）

思考:这个该用哪种触发器？行级触发器

|  |
| --- |
| CREATE OR REPLACE TRIGGER tri\_beforeInsert\_t\_testseq BEFORE INSERT ON t\_testseq FOR EACH ROW BEGIN  SELECT seq\_test.nextval INTO :new.id FROM dual; END;  / |

实现了一个类似mysql的自增长主键的功能.

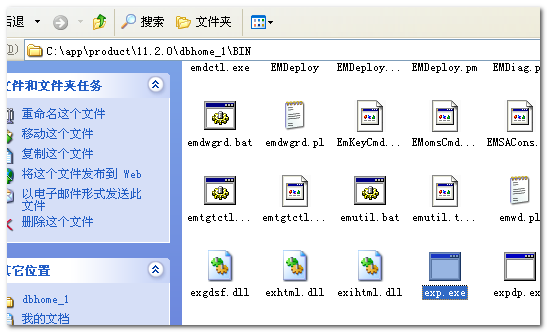
# 数据的备份还原

* 数据库的备份还原
* 数据表的备份还原
* SQL语句的导出。

## 数据库的备份还原-了解

使用oracle自带的备份还原命令：exp(备份),imp（导入）

注意：你的系统中必须有这个命令才能去执行这个命令。



导入和导出命令，既可以直接写命令+参数，也可以使用向导的方式。

下面采用向导的方式：

目标：备份scott用户下的所有对象。

|  |
| --- |
| 每次提取多少数据的时候，当缓存区满了，再写入文件。  导出文件的名字和位置，默认就在当前目录下。  备份的范围，2是默认值，表示备份当前用户下的所有对象。  主要是对象的权限，默认值导出。  导出表对象的时候，是否连数据一起导出，默认值是。  要导出用户确认，直接回车确认  导出的结果：    建议大家在传递该备份的时候，压缩一下： |

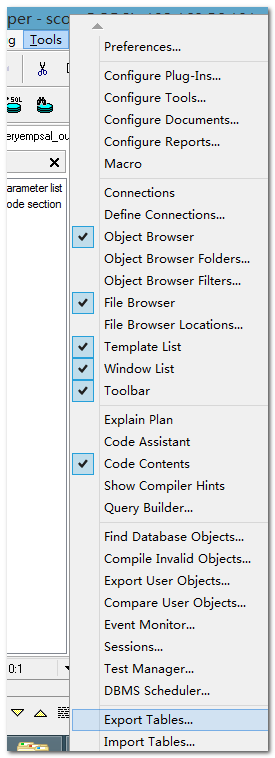
导入：

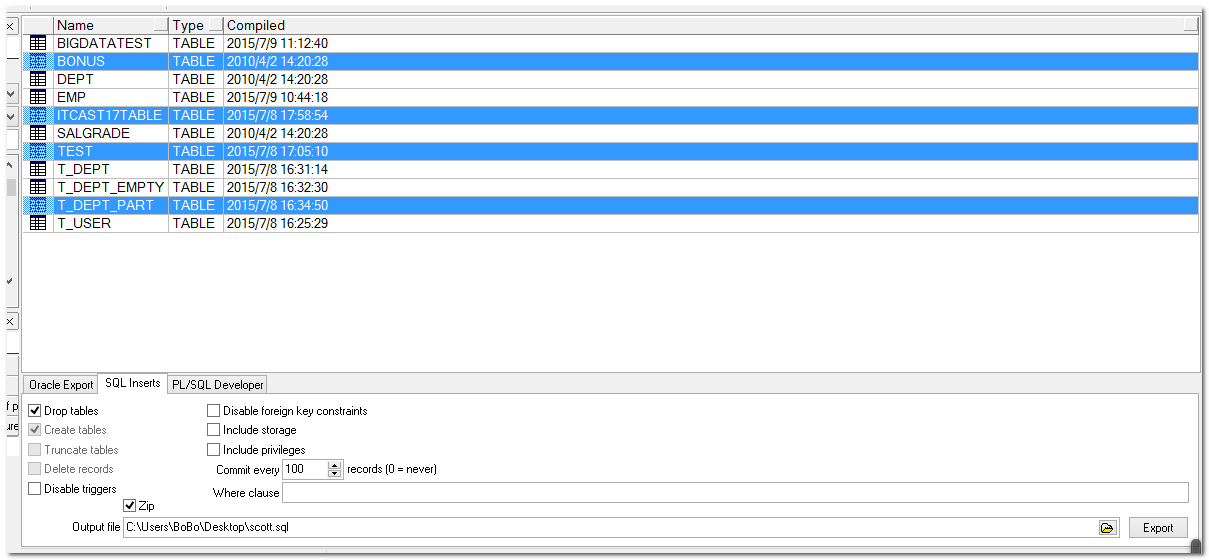
|  |
| --- |
| 新建一个新的用户scott2    要导入的数据的用户名密码。  导入文件的名字和路径，默认在当前文件夹下面。  是否导入选择一些内容。 |

提示：数据库的备份和还原，用于备份和还原数据库的所有的对象的时候。

## 数据表的备份还原

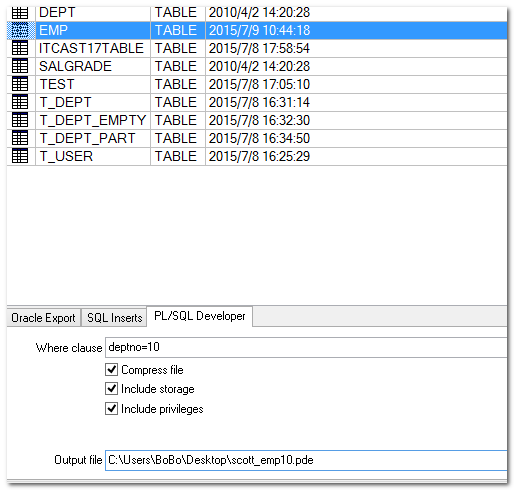
使用plsqlDeveloper工具：

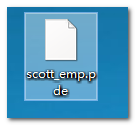




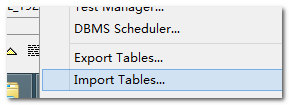
你可以使用ctrl或shilft键进行选择。

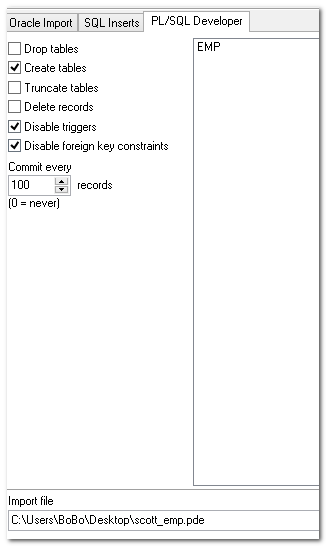
目标：导出emp表中的10号部门的数据





还原方法：





用途：一般如果需要服务器上的某个表的数据，可以用这种方法进行传输。

【导入的前提】

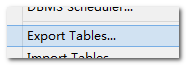
先建立用户，然后在用户下面导入数据。

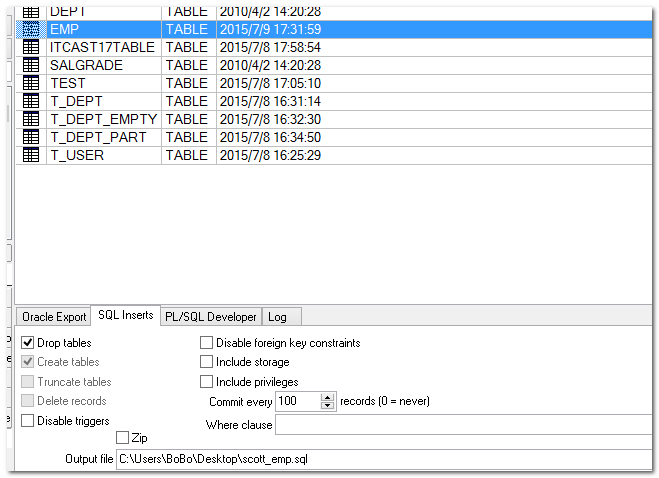
## SQL语句的导出。（表对象转换为SQL）

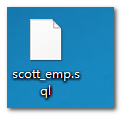
问题来了：

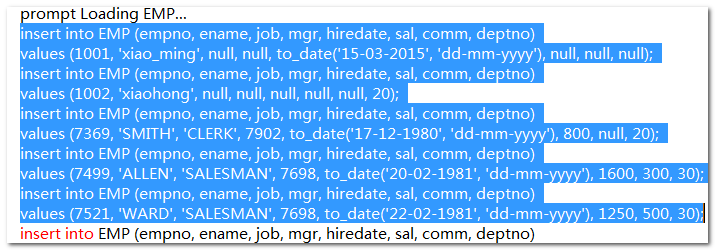
语句怎么弄？如果一条数据，你手动写没问题，那么有100条数据。

想想：起始测试环境下数据库已经有这些数据了，能否将这些数据转换成insert语句。











【扩展】

drop table DEPT cascade constraints;不管dept是否有外表关联，都可以删除。（会自动解除关系）



Oracle重点：

1. 单表的查询（强化）
2. 转换函数（to\_char,to\_date,to\_number）
3. 条件表达式(函数):decode,case when then end;
4. 分页（rownum）--能力
5. 表的连接查询（强化）
6. 高水位（delete和truncation的区别）
7. 序列的创建和使用
8. 用户的创建和权限的配置（配置角色connection,resource------dba---注意hibernate跨域访问会出现问题）
9. 视图作用概念—面试
10. 索引----面试和基于数据库的优化
11. 存储过程—写报表、复杂的业务。java调用。
12. 游标
13. 触发器—面试

[扩展补充:]

1. 降龙十八掌

百度搜索关键

* 表分区的资料(可以将一张表分成很多个表空间来存储,为了提供查询的效率.)—跟分表区分开
* 数据库的Job或schedule
* Dblink---跨数据库访问用的。