触发器和游标

一、触发器

触发器是在事件发生时隐式地自动运行的PL/SQL程序块，不能接收参数，不能被调用。

触发器在数据库里以独立的对象存储，它与存储过程和函数不同的是，存储过程与函数需要用户显示调用才执行，而触发器是由一个事件来启动运行。**即触发器是当某个事件发生时自动地隐式运行。**并且，**触发器不能接收参数。**所以运行触发器就叫触发或点火（firing）。

ORACLE事件指的是对数据库的表进行的INSERT、UPDATE及DELETE操作或对视图进行类似的操作。ORACLE将触发器的功能扩展到了触发ORACLE，如数据库的启动与关闭等。所以触发器常用来完成由数据库的完整性约束难以完成的复杂业务规则的约束，或用来监视对数据库的各种操作，实现审计的功能。

1.1 触发器分类

- DML触发器
- 替代触发器
- 系统触发器

1.2 触发器语法

1.2.1 基本语法

```
CREATE [OR REPLACE] TRIGGER trigger_name
触发时间 {BEFORE | AFTER }
触发事件 {INSERT | DELETE | UPDATE [OF column [, column …]]}
[OR {INSERT | DELETE | UPDATE [OF column [, column …]]}...]
ON 触发对象 [schema.]table_name | [schema.]view_name
[REFERENCING {OLD [AS] old | NEW [AS] new| PARENT as parent}]
[FOR EACH ROW ]
触发条件 [WHEN condition]
DECLARE
  定义变量/常量
BEGIN
  触发操作 PL/SQL_BLOCK | CALL procedure_name;
END;
```

解释说明

- 触发事件
- 引起触发器被触发的事件。 例如：DML语句(INSERT, UPDATE，DELETE语句对表或视图执行数据处理操作)、DDL语句（如CREATE、ALTER、DROP语句在数据库中创建、修改、删除模式对象）、数据库系统事件（如系统启动或退出、异常错误）、用户事件（如登录或退出数据库）。
- 触发时间
- 即该TRIGGER 是在触发事件发生之前（BEFORE）还是之后(AFTER)触发，也就是触发事件和该TRIGGER 的操作顺序。

- 触发操作
- 即该TRIGGER 被触发之后的目的和意图，正是触发器本身要做的事情。 例如：PL/SQL 块。
- 触发对象
- 包括表、视图、模式、数据库。只有在这些对象上发生了符合触发条件的触发事件，才会执行触发操作。
- 触发条件
    - 由WHEN子句指定一个逻辑表达式。只有当该表达式的值为TRUE时，遇到触发事件才会自动执行触发器，使其执行触发操作。
- 触发频率
    - 说明触发器内定义的动作被执行的次数。即语句级(STATEMENT)触发器和行级(ROW)触发器。
        - 语句级(STATEMENT)触发器：是指当某触发事件发生时，该触发器只执行一次；
        - 行级(ROW)触发器：是指当某触发事件发生时，对受到该操作影响的每一行数据，触发器都单独执行一次。

1.2.2 注意

- 触发器不接受参数。
- 一个表上最多可有12个触发器，但同一时间、同一事件、同一类型的触发器只能有一个。并各触发器之间不能有矛盾。
- 在一个表上的触发器越多，对在该表上的DML操作的性能影响就越大。
- 触发器最大为32KB。若确实需要，可以先建立过程，然后在触发器中用CALL语句进行调用。
- **在触发器的执行部分只能用DML语句（SELECT、INSERT、UPDATE、DELETE），不能使用DDL语句（CREATE、ALTER、DROP）。**
- 触发器中不能包含事务控制语句(COMMIT，ROLLBACK，SAVEPOINT)。因为触发器是触发语句的一部分，触发语句被提交、回退时，触发器也被提交、回退了。
- 在触发器主体中调用的任何过程、函数，都不能使用事务控制语句。
- 在触发器主体中不能申明任何Long和blob变量。新值new和旧值old也不能是表中的任何long和blob列。
- 不同类型的触发器(如DML触发器、INSTEAD OF触发器、系统触发器)的语法格式和作用有较大区别。

1.2.3 触发器触发次序

1. 执行 BEFORE语句级触发器;
2. 对与受语句影响的每一行：
    1. 执行 BEFORE行级触发器
    2. 执行 DML语句
    3. 执行 AFTER行级触发器
3. 执行 AFTER语句级触发器

1.3 触发器使用

1.3.1 DML触发器

ORACLE可以在DML语句进行触发，可以在DML操作前或操作后进行触发，并且可以对每个行或语句操作上进行触发。

创建

示例：

```
SELECT * FROM classes;
SELECT * FROM student;
-- 触发器：在特定时间由系统自动出发的事件,执行特定操作
/*
1.不能手动调用;
2.执行顺序与创建顺序无关;
3.多个触发器按顺序全部执行,如果出现系统错误,停止执行;
4.不能包含事物操作,创建时不报错,执行时出错;
*/
/*
CREATE OR REPLACE TRIGGER 触发器的名字
触发时间 [BEFORE/after]
触发事件 [update/delete/insert/UPDATE OR DELETE]
ON 触发对象
declare
   定义变量/常量
BEGIN
  触发操作
END;
*/

CREATE OR REPLACE TRIGGER tri_01
BEFORE
UPDATE
ON classes
DECLARE
   days VARCHAR2(10);
BEGIN
  SELECT to_char(SYSDATE,'day') INTO days FROM dual;
  IF days='星期一' THEN
    raise_application_error(-20001,'星期一不能修改');
  END IF;
END;

-- 多个触发器
CREATE OR REPLACE TRIGGER tri_02
BEFORE
UPDATE
ON classes
BEGIN
  dbms_output.put_line('tri_02');
END;


CREATE OR REPLACE TRIGGER tri_03
BEFORE
UPDATE
ON classes
BEGIN
  raise_application_error(-20003,'星期一不能修改');
  dbms_output.put_line('tri_03');
END;

-- 测试
UPDATE classes SET teacher='T02' WHERE classid='1001';
```

删除

```
-- 删除
DROP TRIGGER tri_01;
DROP TRIGGER tri_02;
DROP TRIGGER tri_03;
```

事务

触发器中不能包含事务操作

```
-- 不能包含事务控制
CREATE OR REPLACE TRIGGER tri_04
BEFORE
UPDATE
ON classes
BEGIN
  COMMIT;
END;
```

条件谓词

当在触发器中包含多个触发事件（INSERT、UPDATE、DELETE）的组合时，为了分别针对不同的事件进行不同的处理，需要使用ORACLE提供的如下条件谓词。

- **INSERTING**：当触发事件是INSERT时，取值为TRUE，否则为FALSE。
- **UPDATING [（column_1,column_2,...,column_x）]**：当触发事件是UPDATE时，如果修改了column_x列，则取值为TRUE，否则为FALSE。其中column_x是可选的。
- **DELETING**：当触发事件是DELETE时，则取值为TRUE，否则为FALSE。

```
-- 条件谓词
CREATE OR REPLACE TRIGGER tri_05
BEFORE
INSERT OR UPDATE OR DELETE
ON classes
BEGIN
  -- 新增时,状态必须是2：未开班
  -- 修改删除时状态必须不能等于0：离校
  IF inserting THEN
    raise_application_error(-20001,'不能进行新增');
  ELSIF updating THEN
    raise_application_error(-20002,'不能进行修改');
  ELSIF deleting THEN
    raise_application_error(-20003,'不能进行删除');
  END IF;
END;

INSERT INTO classes VALUES(1008,'GS490',0,'T04','T01');
```

NEW/OLD

当触发器被触发时，要使用被插入、更新或删除的记录中的列值，有时要使用操作前、后列的值.

- :NEW 修饰符访问操作完成后列的值
- :OLD 修饰符访问操作完成前列的值

特性	INSERT	UPDATE	DELETE
OLD	NULL	实际值	实际值
NEW	实际值	实际值	NULL

示例：

```
-- 对于已毕业的班级信息不能做修改不能删除
CREATE OR REPLACE TRIGGER tri_classes
BEFORE
UPDATE OR DELETE
ON classes
DECLARE
  states classes.state%TYPE;
BEGIN
  SELECT state INTO states WHERE classid=?;
END;

UPDATE classes SET teacher='T02' WHERE classid='1001';

-- insert,update,delete,select
-- :new :insert,update
-- :old :update,delete

-- 修改班级
-- 如果班级状态发生修改
-- 1.0：不能修改
-- 2.1：查询学生信息
CREATE OR REPLACE TRIGGER tri_stu
BEFORE
UPDATE
ON classes
FOR EACH ROW -- 行级操作
DECLARE
  nums NUMBER(10);
BEGIN
  dbms_output.put_line('执行触发器');
  IF :old.state != :new.state THEN
    IF :old.state = 0 THEN
      raise_application_error(-20000,'该班级已毕业，不能修改');
    ELSIF :old.state = 1 THEN
      SELECT COUNT(*) INTO nums FROM student WHERE classid = :new.classid AND
state=1;
      IF nums>0 THEN
        raise_application_error(-20001,'有学生未毕业，不能修改');
      END IF;
    END IF;
  END IF;
END;
```

触发条件

由WHEN子句指定一个逻辑表达式，只允许在行级触发器上指定触发条件，指定UPDATING后面的列的列表。

适当的触发条件可以有效的减少触发器的执行次数。

```
CREATE OR REPLACE TRIGGER tri_stu2
BEFORE
UPDATE
ON classes
FOR EACH ROW -- 行级操作
WHEN(old.state != new.state) -- 触发条件
DECLARE
  nums NUMBER(10);
BEGIN
    dbms_output.put_line('执行触发器');
    IF :old.state = 0 THEN
      raise_application_error(-20000,'该班级已毕业，不能修改');
    ELSIF :old.state = 1 THEN
      SELECT COUNT(*) INTO nums FROM student WHERE classid = :new.classid AND
state=1;
      IF nums>0 THEN
        raise_application_error(-20001,'有学生未毕业，不能修改');
      END IF;
    END IF;
END;

UPDATE classes SET state=1,teacher='T01' WHERE classid=1005;

-- 主键自增
CREATE OR REPLACE TRIGGER tri_teacher
BEFORE
INSERT
ON teacher
FOR EACH ROW
WHEN(new.tid IS NULL)
BEGIN
  :new.tid:='T'||seq_files.nextval;
END;


INSERT INTO classes VALUES(1008,'GS394',1,'T05','T01');
INSERT INTO teacher VALUES(NULL,'admin',1,1);

SELECT 'T'||seq_files.nextval FROM dual;
```

1.3.2 替代触发器

由于在ORACLE里，不能直接对由两个以上的表建立的视图进行操作。所以给出了替代触发器。它就是ORACLE 8专门为进行**视图**操作的一种处理方法。

基本语法:

```
CREATE [OR REPLACE] TRIGGER trigger_name
INSTEAD OF
{INSERT | DELETE | UPDATE [OF column [, column …]]}
[OR {INSERT | DELETE | UPDATE [OF column [, column …]]}...]
ON [schema.] view_name --只能定义在视图上
[REFERENCING {OLD [AS] old | NEW [AS] new| PARENT as parent}]
[FOR EACH ROW ] --因为INSTEAD OF触发器只能在行级上触发,所以没有必要指定
[WHEN condition]
PL/SQL_block | CALL procedure_name;
```

INSTEAD OF 选项使ORACLE激活触发器，而不执行触发事件。只能对视图和对象**视图**建立INSTEAD OF触发器，而不能对表、模式和数据库建立INSTEAD OF 触发器。

示例：

```
CREATE OR REPLACE TRIGGER tri_vstu
INSTEAD OF
UPDATE
ON v_stu
FOR EACH ROW
BEGIN
  dbms_output.put_line('执行触发器');
  IF :old.state=0 THEN
    raise_application_error(-20001,'该学生已经毕业，不能修改信息');
  ELSE

    dbms_output.put_line('可以执行修改');
  END IF;
END;

SELECT * FROM v_stu;
UPDATE v_Stu SET phone=13778659870 WHERE stuno='S001';
COMMIT;
```

创建INSTEAD OF触发器需要注意以下几点：

- 只能被创建在视图上，并且该视图没有指定WITH CHECK OPTION选项。
- 不能指定BEFORE 或 AFTER选项。
- FOR EACH ROW子可是可选的，即INSTEAD OF触发器只能在行级上触发、或只能是行级触发器，没有必要指定。
- 没有必要在针对一个表的视图上创建INSTEAD OF触发器，只要创建DML触发器就可以了。

1.3.3 系统触发器

ORACLE 10G提供的系统事件触发器可以在DDL或数据库系统上被触发。DDL指的是数据定义语言，如CREATE 、ALTER及DROP 等。而数据库系统事件包括数据库服务器的启动或关闭，用户的登录与退出、数据库服务错误等。

创建触发器的一般语法是:

```
CREATE OR REPLACE TRIGGER [sachema.]trigger_name
{BEFORE|AFTER}
{ddl_event_list | database_event_list}
ON { DATABASE | [schema.]SCHEMA }
[WHEN condition]
DECLARE
BEGIN
    PL/SQL_block | CALL procedure_name;
END;
```

下面给出系统触发器的种类和事件出现的时机（前或后）：

事件	允许的时机	说明
STARTUP	AFTER	启动数据库实例之后触发
SHUTDOWN	BEFORE	关闭数据库实例之前触发（非正常关闭不触发）
SERVERERROR	AFTER	数据库服务器发生错误之后触发
LOGON	AFTER	成功登录连接到数据库后触发
LOGOFF	BEFORE	开始断开数据库连接之前触发
CREATE	BEFORE，AFTER	在执行CREATE语句创建数据库对象之前、之后触发
DROP	BEFORE，AFTER	在执行DROP语句删除数据库对象之前、之后触发
ALTER	BEFORE，AFTER	在执行ALTER语句更新数据库对象之前、之后触发
DDL	BEFORE，AFTER	在执行大多数DDL语句之前、之后触发
GRANT	BEFORE，AFTER	执行GRANT语句授予权限之前、之后触发
REVOKE	BEFORE，AFTER	执行REVOKE语句收权限之前、之后触犯发
RENAME	BEFORE，AFTER	执行RENAME语句更改数据库对象名称之前、之后触犯发
AUDIT **/** NOAUDIT	BEFORE，AFTER	执行AUDIT**或**NOAUDIT进行审计或停止审计之前、之后触发

除DML语句的列属性外，其余**事件属性值**可通过调用ORACLE定义的事件属性函数来读取。

函数名称	数据类型	说 明
Ora_sysevent	VARCHAR2（20）	激活触发器的事件名称
Instance_num	NUMBER	数据库实例名
Ora_database_name	VARCHAR2（50）	数据库名称
Server_error(posi)	NUMBER	错误信息栈中posi指定位置中的错误号
Is_servererror(err_number)	BOOLEAN	检查err_number指定的错误号是否在错误信息栈中，如果在则返回TRUE，否则返回FALSE。在触发器内调用此函数可以判断是否发生指定的错误。
Login_user	VARCHAR2(30)	登陆或注销的用户名称

函数名称	数据类型	说 明
Dictionary_obj_type	VARCHAR2(20)	DDL语句所操作的数据库对象类型
Dictionary_obj_name	VARCHAR2(30)	DDL语句所操作的数据库对象名称
Dictionary_obj_owner	VARCHAR2(30)	DDL语句所操作的数据库对象所有者名称
Des_encrypted_password	VARCHAR2(2)	正在创建或修改的经过DES算法加密的用户口令

示例：

创建登录、退出触发器

```
CREATE TABLE log_event(
    user_name VARCHAR2(10),
    address VARCHAR2(20),
    logon_date timestamp,
    logoff_date timestamp
);

--创建登录触发器
CREATE OR REPLACE TRIGGER tr_logon
AFTER
LOGON
ON DATABASE
BEGIN
    INSERT INTO log_event (user_name, address, logon_date)
    VALUES (ora_login_user, ora_client_ip_address, systimestamp);
END tr_logon;

--创建退出触发器
CREATE OR REPLACE TRIGGER tr_logoff
BEFORE
LOGOFF
ON DATABASE
BEGIN
    INSERT INTO log_event (user_name, address, logoff_date)
    VALUES (ora_login_user, ora_client_ip_address, systimestamp);
END tr_logoff;
```

1.3.4 命名块调用

触发器中可以调用其他命名块，因为触发器中不能包含DML语句，所以调用的命名块中也不能包含DML。

```
-- 触发器中可以调用其他命名块，但是命名块中不能包含DML
CREATE OR REPLACE PROCEDURE proc_stu
AS
BEGIN
  INSERT INTO teacher VALUES(NULL,'tom',1,1);
  -- COMMIT;
END;

CALL proc_stu();

SELECT * FROM teacher;
```

```
CREATE OR REPLACE TRIGGER tri_score
BEFORE
INSERT OR UPDATE OR DELETE
ON score
BEGIN
  proc_stu();
END;

SELECT * FROM score;
UPDATE score SET score=99 WHERE stuno='S001' AND cid=2;
```

1.4 触发器和数据字典

相关数据字典：USER_TRIGGERS、ALL_TRIGGERS、DBA_TRIGGERS

```
SELECT TRIGGER_NAME, TRIGGER_TYPE, TRIGGERING_EVENT,TABLE_OWNER,
BASE_OBJECT_TYPE,REFERENCING_NAMES,STATUS, ACTION_TYPE
FROM user_triggers;
```

二、游标

Oracle游标是通过关键字CURSOR的来定义**存储多条查询数据的一种数据结构**，是一组Oracle查询出来的**数据集**，类似数组一样，把查询的数据集存储在内存当中，这些属性包括并发管理、在结果集中的位置、返回的行数，以及是否能够在结果集中向前和/或向后移动（可滚动性）。

游标中有一个 '指针'，从上往下移动（'fetch'），从而**能够遍历每条记录**，通过游标指针可以指向其中一条记录，通过循环游标达到循环数据集的目的。是从表中检索出结果集，从中每次指向一条记录进行交互的机制。

2.1 游标的作用

程序语言是面向记录的，一组变量一次只能存放一个变量或者一条记录，无法直接接收数据库中的查询结果集引入游标就解决了这个问题。

作用

- 指定结果集中特定行的位置。
- 基于当前的结果集位置检索一行或连续的几行。
- 在结果集的当前位置修改行中的数据。
- 对其他用户所做的数据更改定义不同的敏感性级别。
- 可以以编程的方式访问数据库。

2.2 游标的种类

Oracle游标可以分为显式游标和隐式游标两种之分。

显式游标：指的是游标使用之前必须得先声明定义，一般是对查询语句的结果事进行定义游标，然后通过打开游标循环获取结果集内的记录，或者可以根据业务需求跳出循环结束游标的获取。循环完成后，可以通过关闭游标，结果集就不能再获取了。全部操作完全由开发者自己编写完成，自己控制。

隐式游标：指的是PL/SQL自己管理的游标，开发者不能自己控制操作，只能获得它的属性信息。

2.3 游标的使用

2.3.1 显示游标

显式游标在实际开发中经常使用到，可以丰富PL/SQL的开发程序的编写，实现一些循环类的复杂业务。

基本格式:

```
-- 声明游标
CURSOR 名称 IS select语句
-- 打开游标
OPEN 名称
-- 提取数据
FETCH 名称 INTO 变量
-- 关闭游标
CLOSE 名称
```

游标一旦打开后，游标对应的结果集就是静态不会再变了，不管查询的表的基础数据发生了变化。

示例:

```
SELECT * FROM student;

DECLARE
  sturow student%ROWTYPE;
BEGIN
  SELECT * INTO sturow FROM student WHERE stuno='S001';
  dbms_output.put_line(sturow.stuname);
END;

-- 游标：用户处理返回结果是多行的select语句,操作一行数据
/*
CURSOR 名称 IS select语句
OPEN 名称
FETCH 名称 INTO 变量
CLOSE 名称
*/
DECLARE
 CURSOR cs_stu IS SELECT * FROM student;

 sturow student%ROWTYPE;
BEGIN
  -- 判断是否打开游标
  IF cs_stu%ISOPEN THEN
    dbms_output.put_line('打开前');
  END IF;
  -- 打开游标
  OPEN cs_stu;

  IF cs_stu%ISOPEN THEN
    dbms_output.put_line('打开后');
  END IF;
  -- 提取游标数据:一次提取一行
  FETCH cs_stu INTO sturow;
  -- 判断是否提取到数据
```

```
  IF cs_stu%FOUND THEN
    dbms_output.put_line('提取到数据');
  END IF;

  dbms_output.put_line(sturow.stuno||sturow.stuname);
  CLOSE cs_stu;
END;


-- 循环提取所有数据
DECLARE
   CURSOR cs_stu IS SELECT stuname FROM student;
   datas student.stuname%TYPE;-- 存放提取数据
BEGIN
  OPEN cs_stu;

  LOOP
   FETCH cs_stu INTO datas;
   EXIT WHEN cs_stu%NOTFOUND;
   dbms_output.put_line(datas);
  END LOOP;

  CLOSE cs_stu;
END;
```

BULK COLLECT

一次性提取所有数据

```
-- 一次性提取所有数据
DECLARE
   CURSOR cs_stu IS SELECT * FROM student;

   -- table:与游标一起使用时,索引必须使用数字
   TYPE stu_table IS TABLE OF student%ROWTYPE
   INDEX BY BINARY_INTEGER;

   datas stu_table;

   i NUMBER:=0;
BEGIN
  OPEN cs_stu;

  FETCH cs_stu BULK COLLECT INTO datas;-- 提取所有数据保存
  CLOSE cs_stu;

  -- 获取索引表的下标
  dbms_output.put_line(datas.first);
  dbms_output.put_line(datas.last);

  i := datas.first;  -- 1
  WHILE i<=datas.last LOOP  -- i<=17
    dbms_output.put_line(datas(i).stuname);
    i:=i+1;
  END LOOP;
END;
```

```
DECLARE
   CURSOR cs_stu IS SELECT * FROM student;

   -- table:与游标一起使用时,索引必须使用数字
   TYPE stu_table IS TABLE OF student%ROWTYPE
   INDEX BY BINARY_INTEGER;

   datas stu_table;
BEGIN
  OPEN cs_stu;

  FETCH cs_stu BULK COLLECT INTO datas;-- 提取所有数据保存
  CLOSE cs_stu;

  FOR i IN datas.first..datas.last LOOP
    dbms_output.put_line(datas(i).stuname);
  END LOOP;
END;
```

带参数的游标

```
-- 带参数
-- 定义时定义参数(不带长度);打开时传递参数值
DECLARE
   CURSOR cs_stu(cid VARCHAR2) IS SELECT * FROM student WHERE classid=cid;

   -- table:与游标一起使用时,索引必须使用数字
   TYPE stu_table IS TABLE OF student%ROWTYPE
   INDEX BY BINARY_INTEGER;

   datas stu_table;
BEGIN
  OPEN cs_stu('1005');

  FETCH cs_stu BULK COLLECT INTO datas;-- 提取所有数据保存
  CLOSE cs_stu;

  FOR i IN datas.first..datas.last LOOP
    dbms_output.put_line(datas(i).stuname);
  END LOOP;
END;

-- 根据老师查询该老师所带的班级
SELECT posts FROM teacher WHERE tid='T04';
-- posts=2
SELECT * FROM classes WHERE teacher='T01';
-- posts=1
SELECT * FROM classes WHERE headmaster='T04';
```

游标变量

游标在使用时才确定结果集，定义时查询语句不确定，可以使用游标变量实现这样的效果

```
-- 游标变量：游标在使用时才确定结果集，定义时查询语句不确定
DECLARE
```

```
  -- 定义自定义的游标类型
  TYPE CURSOR_CLS IS REF CURSOR;

  CS_CLASS CURSOR_CLS;

  -- 定义变量,获取教师职位
  TPOSTS TEACHER.POSTS%TYPE;
  -- 定义变量,输入教师的编号
  TIDS TEACHER.TID%TYPE;

  -- 接收提取结果的变量
  TYPE CLS_TABLE IS TABLE OF CLASSES%ROWTYPE INDEX BY BINARY_INTEGER;
  DATAS CLS_TABLE;
BEGIN
  -- 1.输入教室编号，查询教师职位
  TIDS := &TIDS;
  SELECT POSTS INTO TPOSTS FROM TEACHER WHERE TID = TIDS;

  -- 2.根据职位判断游标提取的结果集:select语句
  IF TPOSTS = '2' THEN
    DBMS_OUTPUT.PUT_LINE('教员');
    OPEN CS_CLASS FOR
      SELECT * FROM CLASSES WHERE TEACHER = TIDS;
  ELSIF TPOSTS = '1' THEN
    DBMS_OUTPUT.PUT_LINE('班主任');
    OPEN CS_CLASS FOR
      SELECT * FROM CLASSES WHERE HEADMASTER = TIDS;
  ELSE
    DBMS_OUTPUT.PUT_LINE('未定义该老师的职位');
  END IF;

  -- 3.提取游标数据
  FETCH CS_CLASS BULK COLLECT
    INTO DATAS;
  -- 4.关闭游标
  CLOSE CS_CLASS;

  -- 5.打印结果
  FOR I IN DATAS.FIRST .. DATAS.LAST LOOP
    DBMS_OUTPUT.PUT_LINE(DATAS(I).CLASSNAME);
  END LOOP;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('查询的老师不存在');
END;
```

for循环使用

可以使用for循环简化游标操作

```
-- for简化游标：不需要打开，提取，关闭游标
-- for in 游标明
DECLARE
   CURSOR cs_stu IS SELECT * FROM student;
BEGIN
```

```
  -- for 循环变量 in 游标名 loop .. end loop;
  -- 循环变量:获取游标提取到的一行数据
  FOR i IN cs_stu LOOP
    DBMS_OUTPUT.PUT_LINE('第'||cs_stu%ROWCOUNT||'个学生：');
    DBMS_OUTPUT.PUT_LINE('姓名:'||i.stuname||'  身份证号：'||i.idcard);
  END LOOP;
END;


DECLARE
  TYPE cursor_s IS REF CURSOR;
  cs_stu cursor_s;

  datas student%ROWTYPE;
BEGIN
  -- for 循环变量 in 游标名 loop .. end loop;
  -- 循环变量:获取游标提取到的一行数据
  OPEN cs_stu FOR SELECT * FROM student;
  FETCH cs_stu INTO datas;
  CLOSE cs_stu;
  DBMS_OUTPUT.PUT_LINE('姓名:'||datas.stuname||'  身份证号：'||datas.idcard);
  /*
  FOR i IN cs_stu LOOP
    DBMS_OUTPUT.PUT_LINE('第'||cs_stu%ROWCOUNT||'个学生：');
    DBMS_OUTPUT.PUT_LINE('姓名:'||i.stuname||'  身份证号：'||i.idcard);
  END LOOP;
  */
END;


BEGIN
  FOR i IN (SELECT * FROM student) LOOP
    DBMS_OUTPUT.PUT_LINE('姓名:'||i.stuname||'  身份证号：'||i.idcard);
  END LOOP;
END;
```

2.3.2 隐式游标

隐式游标虽然不能像显式游标一样具有操作性，但是在实际开发过程当中还是经常使用到它的属性值，隐式游标主要是用在select语句活DML语句时，PL/SQL程序会自动打开隐式游标，这个隐式游标是不受开发者控制的。

常用隐式游标

属性	返回值类型	作用
sql%isopen	布尔型	判断游标是否 '开启'
sql%found	布尔型	判断游标是否 '获取' 到值
sql%notfound	布尔型	判断游标是否 '没有获取' 到值（常用于 "退出循环"）
sql%rowcount	布尔型	'当前' 成功执行的数据行数（非 "总记录数"）

```
-- 隐式游标：SQL
DECLARE
   datas student%ROWTYPE;
```

```
BEGIN
  SELECT * INTO datas FROM student WHERE stuno='1111';
  EXCEPTION
    WHEN no_data_found THEN
      DBMS_OUTPUT.PUT_LINE('没有查询结果');
    WHEN too_many_rows THEN
       DBMS_OUTPUT.PUT_LINE('查询结果不唯一');
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('错误');
END;

DECLARE
   datas student%ROWTYPE;
BEGIN
  UPDATE student SET stuname='张三';

  DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT);
  IF SQL%NOTFOUND THEN
     DBMS_OUTPUT.PUT_LINE('修改的数据不存在');
  ELSIF SQL%FOUND THEN
     DBMS_OUTPUT.PUT_LINE('修改成功');
     ROLLBACK;
  END IF;
END;

SELECT * FROM student WHERE stuno='S001';
```

作业

1. 创建系统触发器，每次创建新对象时，记录创建的相关信息，记录到表中（操作人，操作时间，操作，对象类型，对象的名称，对象的创建者）；
2. 创建系统触发器，每次删除对象时，记录相关信息，记录到表中；
3. 创建触发器，登录/退出，记录数据库的访问情况；
4. 游标
    1. 查询学生表，循环输出所有信息；
    2. 查询教师表，一次提取所有数据并输出；