ch19. 저장 서브프로그램

19-1. 저장 서브프로그램

> 저장 서브프로그램이란?

- 익명 블록 *anonymous block* : 이름이 정해져 있지 않은 PL/SQL블록
- 익명 블록은 오라클에 저장되지 않기 때문에 한 번 실행한 뒤에 다시 실행 하려면 다시 작성하여 실행해야 한다.
- 그런데, PL/SQL로 만든 프로그램을 주기적으로 또는 필요할 때마다 여러 번 사용해야 하는 상황이 빈번히 발생.
    - 이럴 경우, 프로그램을 오라클에 저장해 두고 실행 가능하다.
- 저장 서브프로그램 *stored subprogram* : 여러 번 사용할 목적으로 이름을 지정하여 오라클에

  저장해 두는 PL/SQL 프로그램
- 오라클에서의 저장 서브프로그램 구현 방식
    - 프로시저, 함수, 패키지, 트리거
        - 저장 프로시저 : 일반적으로 특정 처리 작업 수행을 위한 서브 프로그램으로

          SQL문에서는 사용할수 없다.
        - 저장 함수 : 일반적으로 특정 연산을 거친 결과 값을 반환하는 서브프로그램으로

          SQL문에서 사용할 수 있다.
        - 패키지 : 저장 서브프로그램을 그룹화하는 데 사용한다.
        - 트리거 : 특정 상황 (이벤트) 이 발생할 때 자동으로 연달아 수행할 기능을 구현하는 데

          사용한다.

19-2. 프로시저

저장 프로시저는 특정 처리 작업을 수행하는 데 사용하는 저장 서브프로그램으로 용도에 따라 파라미터를 사용할 수 있고 사용하지 않을 수도 있다.

파라미터를 사용하지 않는 프로시저

프로시저 생성하기

- 작업 수행에 별다른 입력 데이터가 필요하지 않을 경우에 파라미터를 사용하지 않는 프로시저를 사용한다.

```
CREATE [OR REPLACE] PROCEDURE 프로시저 이름
IS | AS
   선언부
BEGIN
  실행부
EXCEPTION
  예외 처리부
END [프로시저 이름];
```

[OR REPLACE]	지정한 프로시저 이름을 가진 프로시저가 이미 존재하는 경우에 현재 작성한 내용으로 대체. 즉 덮어쓴다는 뜻이며 생략 가능한 옵션이다.
프로시저 이름	저장할 프로시저의 고유 이름을 지정. 같은 스키마 내에서 중복될 수 없다.
IS | AS	선언부를 시작하기 위해 IS 또는 AS 키워드를 사용. 선언부가 존재하지 않더라도 반드시 명시. DECLARE 키워드는 사용하지 않는다.
EXCEPTION	예외 처리부는 생략 가능.
END	프로시저 생성의 종료를 뜻하며 프로시저 이름은 생략 가능.

- 프로시저 생성하기

```
CREATE OR REPLACE PROCEDURE pro_noparam
IS
   V_EMPNO NUMBER(4) := 7788;
   V_ENAME VARCHAR2(10);
BEGIN
   V_ENAME := 'SCOTT';
   DBMS_OUTPUT.PUT_LINE('V_EMPNO : ' || V_EMPNO);
   DBMS_OUTPUT.PUT_LINE('V_ENAME : ' || V_ENAME);
END;
/
```

SQL*PLUS로 프로시저 실행하기

```
EXECUTE 프로시저 이름;
```

- 생성한 프로시저 실행하기

```
$ SET SERVEROUTPUT ON;
$ EXECUTE pro_noparam;
```

PL/SQL 블록에서 프로시저 실행하기

```
BEGIN
  프로시저 이름;
END;
```

- 익명 블록에서 프로시저 실행하기

```
BEGIN
   pro_noparam;
END;
/
```

프로시저 내용 확인하기

- 이미 저장되어 있는 프로시저를 포함하여 서브프로그램의 소스 코드 내용을 확인하려면 USER_SOURCE 데이터 사전에서 조회한다.

USER_SOURCE 의 열	설명
NAME	서브 프로그램(생성 객체) 이름
TYPE	서브 프로그램 종류 (PROCEDURE, FUNCTION 등)
LINE	서브프로그램에 작성한 줄 번호
TEXT	서브프로그램에 작성한 소스 코드

- USER_SOURCE를 통해 프로시저 확인하기(오라클)

```
SELECT *
  FROM USER_SOURCE
 WHERE NAME = 'PRO_NOPARAM';
```

- USER_SOURCE를 통해 프로시저 확인하기(SQL*PLUS)

```
$ SELECT TEXT
    FROM USER_SOURCE
   WHERE NAME = 'PRO_NOPARAM';
```

프로시저 삭제하기

```
$ DROP PROCEDURE PRO_NOPARAM;
```

파라미터를 사용하는 프로시저

- 프로시저를 실행하기 위해 입력 데이터가 필요한 경우에 파라미터를 정의할 수 있다.
  - 파라미터는 여러 개 작성할 수 있다.

```
CREATE [OR REPLACE] PROCEDURE 프로시저 이름

[(파라미터 이름1 [modes] 자료형 [ := | DEFAULT 기본값],
  파라미터 이름2 [modes] 자료형 [ := | DEFAULT 기본값],
   ...
  파라미터 이름2 [modes] 자료형 [ := | DEFAULT 기본값]
)]
IS | AS
   선언부
BEGIN
  실행부
EXCEPTION
  예외 처리부
END [프로시저 이름];
```

- 파라미터를 지정할 때 사용하는 모드

IN	지정하지 않으면 기본값으로 프로시저를 호출할 때 값을 입력받는다.
OUT	호출할 때 값을 반환한다.
IN OUT	호출할 때 값을 입력받은 후 실행 결과 값을 반환한다.

IN 모드 파라미터

- 프로시저에 파라미터 지정하기

```
CREATE OR REPLACE PROCEDURE pro_param_in
(
   param1 IN NUMBER,
   param2 NUMBER,
   param3 NUMBER := 3,
   param4 NUMBER DEFAULT 4
)
IS

BEGIN
   DBMS_OUTPUT.PUT_LINE('param1 : ' || param1);
   DBMS_OUTPUT.PUT_LINE('param2 : ' || param2);
   DBMS_OUTPUT.PUT_LINE('param3 : ' || param3);
   DBMS_OUTPUT.PUT_LINE('param4 : ' || param4);
END;
/
```

- 파라미터를 입력하여 프로시저 사용하기

```
EXECUTE pro_param_in(1,2,9,8);
```

- 기본값이 지정된 파라미터 입력을 제외하고 프로시저 사용하기

```
EXECUTE pro_param_in(1, 2);
```

- 실행에 필요한 개수보다 적은 파라미터를 입력하여 프로시저 실행하기

```
EXECUTE pro_param_in(1);
```

- 파라미터 이름을 활용하여 프로시저에 값 입력하기

```
EXECUTE pro_param_in(param1 => 10, param2 => 20);
```

- 파라미터 값을 지정할 때는 다음 세 가지 지정 방식을 사용할 수 있다.

종류	설명
위치 지정	지정한 파라미터 순서대로 값을 지정하는 방식
이름 지정	=> 연산자로 파라미터 이름을 명시하여 값을 지정하는 방식
혼합 지정	일부 파라미터는 순서대로 값만 지정하고 일부 파라미터는 => 연산자로 값을 지정하는 방식

OUT 모드 파라미터

OUT 모드를 사용한 파라미터는 프로시저 실행 후 호출한 프로그램으로 값을 반환한다.

- OUT 모드 파라미터 정의하기

```
CREATE OR REPLACE PROCEDURE pro_param_out
(
   in_empno IN EMP.EMPNO%TYPE,
   out_ename OUT EMP.ENAME%TYPE,
   out_sal OUT EMP.SAL%TYPE
)
IS

BEGIN
   SELECT ENAME, SAL INTO out_ename, out_sal
     FROM EMP
    WHERE EMPNO = in_empno;
END pro_param_out;
/
```

- OUT 모드 파라미터 사용하기

```
DECLARE
   v_ename EMP.ENAME%TYPE;
   v_sal EMP.SAL%TYPE;
BEGIN
   pro_param_out(7788, v_ename, v_sal);
   DBMS_OUTPUT.PUT_LINE('ENAME : ' || v_ename);
   DBMS_OUTPUT.PUT_LINE('SAL : ' || v_sal);
END;
/
```

IN OUT 모드 파라미터

IN OUT 모드로 선언한 파라미터는 IN, OUT 으로 선언한 파라미터 기능을 동시에 수행한다 . 즉 , 값을 입력받을 때와 프로시저 수행 후 결과값을 반환 할 때 사용한다.

- IN OUT 모드 파라미터 정의하기

```
CREATE OR REPLACE PROCEDURE pro_param_inout
(
   inout_no IN OUT NUMBER
)
IS

BEGIN
   inout_no := inout_no * 2;
END pro_param_inout;
/
```

- IN OUT 모드 파라미터 사용하기

```
DECLARE
   no NUMBER;
BEGIN
   no := 5;
   pro_param_inout(no);
   DBMS_OUTPUT.PUT_LINE('no : ' || no);
END;
/
```

프로시저 오류 정보 확인하기

- 다음 방법은 다른 서브프로그램의 오류에도 똑같이 적용할 수 있다.

- 생성할 때 오류가 발생하는 프로시저 알아보기

```
CREATE OR REPLACE PROCEDURE pro_err
IS
   err_no NUMBER;
BEGIN
   err_no = 100;
   DBMS_OUTPUT.PUT_LINE('err_no : ' || err_no);
```

```
END pro_err;
/
```

⇒ 서브프로그램을 만들 때 발생한 요류는 SHOW ERRORS명령어와 USER_ERRORS 데이터 사전을
조회하여 확인할 수 있다.

SHOW ERRORS로 오류 확인

```
SHOW ERRORS;
```

- 만약 최근에 발생한 프로그램 오류가 아니라 특정 프로그램의 오류 정보를 확인하고 싶다면
  프로그램 종류와 이름을 추가로 지정하면 된다.

  ```
  SHOW ERR 프로그램종류 프로그램이름;
  SHOW ERR PROCEDURE pro_err;
  ```

USER_ERRORS로 오류 확인하기

```
SELECT *
  FROM USER_ERRORS
 WHERE NAME = 'PRO_ERR';
```

19-3. 함수

함수 생성하기

함수는 반환 값의 자료형과 실행부에서 반환할 값을 RETURN절 및 RETURN문으로 명시해야 한다.

실행부의 RETURN문이 실행되면 함수 실행은 즉시 종료된다.

- 함수 생성하기

```
CREATE OR REPLACE FUNCTION func_aftertax(
   sal IN NUMBER
)
RETURN NUMBER
IS
   tax NUMBER := 0.05;
BEGIN
   RETURN (ROUND(sal - (sal * tax)));
END func_aftertax;
/
```

함수 실행하기

- PL/SQL에서 함수 실행하기

```
DECLARE
   aftertax NUMBER;
BEGIN
   aftertax := func_aftertax(3000);
   DBMS_OUTPUT.PUT_LINE('after-tax income : ' || aftertax);
END;
/
```

- SQL문에서 함수 사용하기

```
SELECT func_aftertax(3000)
  FROM DUAL;
```

- 함수에 테이블 데이터 사용하기

```
SELECT EMPNO, ENAME, SAL, func_aftertax(SAL) AS AFTERTAX
  FROM EMP;
```

함수 삭제하기

```
DROP FUNCTION func_aftertax;
```

19-4. 패키지

- 패키지는 업무나 기능 면에서 연관성이 높은 프로시저, 함수 등 여러 개의 PL/SQL 서브 프로그램을 하나의 논리 그룹으로 묶어 통합, 관리하는 데 사용하는 객체.
- 프로시저나 함수 등은 각각 개별 기능을 수행하기 위해 제작 후 다로 저장했다.
- 패키지를 사용하여 서브프로그램을 그룹화할 때 장점

장점	설명
모듈성	서브프로그램을 포함한 여러 PL/SQL 구성 요소를 모듈화할 수 있다, 모듈성은 잘 묶어 둔다는 뜻으로 프로그램의 이해를 쉽게 하고 패키지 사이의 상호 작용을 더 간편하고 명료하게 해주는 역할을 한다. 즉 PL/SQL로 제작한 프로그램의 사용 및 관리에 큰 도움을 준다.
쉬운 응용 프로그램 설계	패키지에 포함할 서브프로그램은 완벽하게 완성되지 않아도 정의가 가능하다. 이 때문에 전체 소스 코드를 다 작성하기 전에 미리 패키지에 저장할 서브프로그램을 지정할 수 있으므로 설계가 수월해진다.
정보 은닉	제작 방식에 따라 패키지에 포함하는 서브프로그램의 외부 노출 여부 또는 접근 여부를 지정할 수 있다. 즉 서브프로그램을 사용할 때 보안을 강화할 수 있다.
기능성 향상	V ㅐ 키지 내부에는 서브프로그램 외에 변수, 커서 ,예외 등도 각 세션이 유지되는 동안 선언해서 공용으로 사용할 수 있다. 예를 들어 특정 커서 데이터는 세션이 종료되기 전까지 보존되므로 여러 서브프로그램에서 사용할 수 있다.
성능 향상	패키지를 사용할 때 페키지에 포함한 모든 서브 프로그램이 메모리에 한번에 로딩되는데 메모리에 로딩된 후의 호출은 디스크 I/O를 일으키지 않으므로 성능이 향상된다.

패키지 구조와 생성

- 패키지 생성하기

```
CREATE OR REPLACE PACKAGE pkg_example
IS
   spec_no NUMBER := 10;
```

```
   FUNCTION func_aftertax(sal NUMBER) RETURN NUMBER;
   PROCEDURE pro_emp(in_empno IN EMP.EMPNO%TYPE);
   PROCEDURE pro_dept(in_deptno IN DEPT.DEPTNO%TYPE);
END;
/
```

- 패키지 명세 확인하기

```
SELECT TEXT
  FROM USER_SOURCE
 WHERE TYPE = 'PACKAGE'
   AND NAME = 'PKG_EXAMPLE';
```

- 패키지 명세 확인하기

```
DESC pkg_example;
```

- 패키지 본문 생성하기

```
CREATE OR REPLACE PACKAGE BODY pkg_example
IS
   body_no NUMBER := 10;

   FUNCTION func_aftertax(sal NUMBER) RETURN NUMBER
      IS
         tax NUMBER := 0.05;
      BEGIN
         RETURN (ROUND(sal - (sal * tax)));
   END func_aftertax;

   PROCEDURE pro_emp(in_empno IN EMP.EMPNO%TYPE)
      IS
         out_ename EMP.ENAME%TYPE;
         out_sal EMP.SAL%TYPE;
      BEGIN
         SELECT ENAME, SAL INTO out_ename, out_sal
           FROM EMP
          WHERE EMPNO = in_empno;

         DBMS_OUTPUT.PUT_LINE('ENAME : ' || out_ename);
         DBMS_OUTPUT.PUT_LINE('SAL : ' || out_sal);
   END pro_emp;

PROCEDURE pro_dept(in_deptno IN DEPT.DEPTNO%TYPE)
   IS
      out_dname DEPT.DNAME%TYPE;
```

```
      out_loc DEPT.LOC%TYPE;
   BEGIN
      SELECT DNAME, LOC INTO out_dname, out_loc
        FROM DEPT
       WHERE DEPTNO = in_deptno;

      DBMS_OUTPUT.PUT_LINE('DNAME : ' || out_dname);
      DBMS_OUTPUT.PUT_LINE('LOC : ' || out_loc);
   END pro_dept;
END;
/
```

- 프로시저 오버로드하기

```
CREATE OR REPLACE PACKAGE pkg_overload
IS
   PROCEDURE pro_emp(in_empno IN EMP.EMPNO%TYPE);
   PROCEDURE pro_emp(in_ename IN EMP.ENAME%TYPE);
END;
/
```

- 패키지 본문에서 오버로드된 프로시저 작성하기

```
CREATE OR REPLACE PACKAGE BODY pkg_overload
IS
   PROCEDURE pro_emp(in_empno IN EMP.EMPNO%TYPE)
      IS
         out_ename EMP.ENAME%TYPE;
         out_sal EMP.SAL%TYPE;
      BEGIN
         SELECT ENAME, SAL INTO out_ename, out_sal
           FROM EMP
          WHERE EMPNO = in_empno;

         DBMS_OUTPUT.PUT_LINE('ENAME : ' || out_ename);
         DBMS_OUTPUT.PUT_LINE('SAL : ' || out_sal);
      END pro_emp;

   PROCEDURE pro_emp(in_ename IN EMP.ENAME%TYPE)
      IS
         out_ename EMP.ENAME%TYPE;
         out_sal EMP.SAL%TYPE;
      BEGIN
         SELECT ENAME, SAL INTO out_ename, out_sal
           FROM EMP
          WHERE ENAME = in_ename;

         DBMS_OUTPUT.PUT_LINE('ENAME : ' || out_ename);
         DBMS_OUTPUT.PUT_LINE('SAL : ' || out_sal);
      END pro_emp;
```

```
END;
/
```

패키지 사용하기

- 패키지에 포함된 서브 프로그램 실행하기

```
BEGIN
   DBMS_OUTPUT.PUT_LINE('--pkg_example.func_aftertax(3000)--');
   DBMS_OUTPUT.PUT_LINE('after-tax:' || pkg_example.func_aftertax(3000));

   DBMS_OUTPUT.PUT_LINE('--pkg_example.pro_emp(7788)--');
   pkg_example.pro_emp(7788);

   DBMS_OUTPUT.PUT_LINE('--pkg_example.pro_dept(10)--' );
   pkg_example.pro_dept(10);

   DBMS_OUTPUT.PUT_LINE('--pkg_overload.pro_emp(7788)--' );
   pkg_overload.pro_emp(7788);

   DBMS_OUTPUT.PUT_LINE('--pkg_overload.pro_emp(''SCOTT'')--' );
   pkg_overload.pro_emp('SCOTT');
END;
/
```

패키지 삭제하기

```
패키지 명세와 본문을 한번에 삭제하기
DROP PACKAGE 패키지 이름;

패키지의 본문만을 삭제하기
DROP PACKAGE BODY 패키지 이름;
```

19-5. 트리거

트리거란?

오라클에서 트리거는 데이터 베이스 안의 특정 상황이나 동작, 즉 이벤트가 발생할 경우 자동으로 실행되는 기능을 정의하는 PL/SQL 서브프로그램이다.

- 트리거의 장점
  1. 데이터와 연관된 여러 작업을 수행하기 위해 여러 PL/SQL문 또는 서브프로그램을 일일이 실행해야 하는 번거로움을 줄일 수 있다. 즉 데이터 관련 작업을 좀 더 간편하게 수행할 수 있다.
  2. 제약 조건 constraints 만으로 구현이 어렵거나 불가능한 좀 더 복잡한 데이터 규칙을 정할 수 있어 더 수준 높은 데이터 정의가 가능하다.
  3. 데이터 변경과 관련된 일련의 정보를 기록해 둘 수 있으므로 여러 사용자가 공유하는 데이터 보안성과 안정성 그리고 문제가 발생했을 때 대처 능력을 높일 수 있다.

- 하지만 트리거는 특정 작업 또는 이벤트 발생으로 다른 데이터 작업을 추가로 실행하기 때문에 무분별하게 사용허면 데이터베이스 성능을 떨어뜨리는 원인이 되므로 주의가 필요하다.
- 트리거는 테이블, 뷰, 스키마, 데이터베이스 수준에서 다음과 같은 이벤트에 동작을 지정할 수 있다.
  - 데이터 조작어(DML): INSERT, UPDATE, DELETE
  - 데이터 정의어(DDL): CREATE, ALTER, DROP
  - 데이터베이스 동작: SERVERERROR, LOGON, LOGOFF, STARTUP, SHUTDOWN
- 트리거가 발생할 수 있는 이벤트 종류에 따라 오라클을 트리거를 다음과 같이 구분한다.

종류	설명
DML 트리거	INSERT, UPDATE, DELETE와 같은 DML 명령어를 기점으로 동작함
DDL 트리거	CREATE, ALTER, DROP과 같은 DDL 명령어를 기점으로 동작함
INSTEAD OF 트리거	뷰(View)에 사용하는 DML 명령어를 기점으로 동작함
시스템(system) 트리거	데이터베이스나 스키마 이벤트로 동작함

종류	설명
단순(simple) 트리거	다음 각 시점(timing point)에 동작함- 트리거를 작동시킬 문장이 실행되기 전 시점- 트리거를 작동시킬 문장이 실행된 후 시점- 트리거를 작동시킬 문장이 행에 영향을 미치기 전 시점- 트리거를 작동시킬 문장이 행에 영향을 준 후 시점
복합(compund) 트리거	단순 트리거의 여러 시점에 동작함

DML 트리거

DML 트리거 형식

- DML 트리거는 특정 테이블에 DML 명령어를 실행했을 때 작동하는 트리거.

```
CREATED [OR REPLACE] TRUGGER 트리거 이름
BEFORE | AFTER
INSERT | UPDATE | DELETE ON 테이블 이름
REFERENCING OLD as old | New as new
FOR EACH ROW WHEN 조건식
FOLLOWS 트리거 이름2, 트리거 이름3, ...
ENABLE | DISABLE

DECLARE
  선언부
BEGIN
  실행부
EXCEPTION
  예외 처리부
END;
```

DML 트리거의 제작 및 사용 (BEFORE)

트리거를 적용할 테이블을 EMP 테이블을 복사하여 생성한다.

- EMP_TRG 테이블 생성하기

```
CREATE TABLE EMP_TRG
    AS SELECT * FROM EMP;
```

trg_emp_nodml_weekend 트리거 생성: 주말에 EMP_TGR 테이블에 DML 명령어를 사용하면 오류를 일으키고, DML 명령어 실행을 취소한다.

- DML 실행 전에 수행할 트리거 생성하기

```
CREATE OR REPLACE TRIGGER trg_emp_nodml_weekend
BEFORE
INSERT OR UPDATE OR DELETE ON EMP_TRG
BEGIN
   IF TO_CHAR(sysdate, 'DY') IN ('토', '일') THEN
      IF INSERTING THEN
         raise_application_error(-20000, '주말 사원정보 추가 불가');
      ELSIF UPDATING THEN
         raise_application_error(-20001, '주말 사원정보 수정 불가');
      ELSIF DELETING THEN
         raise_application_error(-20002, '주말 사원정보 삭제 불가');
      ELSE
         raise_application_error(-20003, '주말 사원정보 변경 불가');
      END IF;
   END IF;
END;
/
```

트리거는 특정 이벤트가 발생할 때 자동으로 작동하는 서브프로그램이므로, 프로시저나 함수와 같이 EXECUTE 또는 PL/SQL 블록에서 따로 실행하지 못한다.

- 평일 날짜로 EMP_TRG 테이블 UPDATE하기

```
UPDATE emp_trg SET sal = 3500 WHERE empno = 7788;
```

- 주말 날짜에 EMP_TRG 테아블 UPDATE하기
  - 주말에 DML 명령어가 실행되면 오류를 발생시켜, 명령어가 취소되게 된다.

```
UPDATE emp_trg SET sal = 3500 WHERE empno = 7788;
```

DML 트리거의 제작 및 사용(AFTER)

- DML 명령어가 실행된 후 작동하는 AFTER 트리거 제작하기
- 앞에서 생성한 EMP_TRG 테이블에 DML 명령어가 실행되었을 때 테이블에 수행된 DML 명령어의 종류, DML을 실행시킨 사용자, DML 명령어가 수행된 날짜와 시간을 저장할 EMP_TRG_LOG 테이블 생성하기

```
CREATE TABLE EMP_TRG_LOG(
   TABLENAME VARCHAR2(10), -- DML이 수행된 테이블 이름
   DML_TYPE VARCHAR2(10),  -- DML 명령어의 종류
   EMPNO NUMBER(4),        -- DML 대상이 된 사원 번호
   USER_NAME VARCHAR2(30), -- DML을 수행한 USER 이름
   CHANGE_DATE DATE        -- DML이 수행된 날짜
);
```

- DML 실행 후 수행할 트리거 생성하기

```
CREATE OR REPLACE TRIGGER trg_emp_log
AFTER
INSERT OR UPDATE OR DELETE ON EMP_TRG
FOR EACH ROW

BEGIN

   IF INSERTING THEN
      INSERT INTO emp_trg_log
      VALUES ('EMP_TRG', 'INSERT', :new.empno,
               SYS_CONTEXT('USERENV', 'SESSION_USER'), sysdate);

   ELSIF UPDATING THEN
      INSERT INTO emp_trg_log
      VALUES ('EMP_TRG', 'UPDATE', :old.empno,
               SYS_CONTEXT('USERENV', 'SESSION_USER'), sysdate);

   ELSIF DELETING THEN
      INSERT INTO emp_trg_log
      VALUES ('EMP_TRG', 'DELETE', :old.empno,
               SYS_CONTEXT('USERENV', 'SESSION_USER'), sysdate);
   END IF;
END;
/
```

- EMP_TRG 테이블에 INSERT 실행하기

```
INSERT INTO EMP_TRG
VALUES(9999, 'TestEmp', 'CLERK', 7788,
       TO_DATE('2018-03-03', 'YYYY-MM-DD'), 1200, null, 20);
```

- EMP_TRG 테이블에 INSERT 실행하기(COMMIT하기)

```
COMMIT;
```

- EMP_TRG 테이블의 INSERT 확인하기

```
SELECT *
  FROM EMP_TRG;
```

- EMP_TRG_LOG 테이블의 INSERT 기록 확인하기

```
SELECT *
  FROM EMP_TRG_LOG;
```

- EMP_TRG 테이블에 UPDATE 실행하기

```
UPDATE EMP_TRG
   SET SAL = 1300
 WHERE MGR = 7788;
```

- EMP_TRG 테이블에 UPDATE 실행하기(COMMIT하기)
  - 두 개 행이 DML문에 영향을 받았으므로 트리거에 지정한 FOR EACH ROW 옵션으로 트리거는 두 번 실행한다.

```
COMMIT;
```

트리거 관리

트리거 정보 조회

- 트리거 정보를 확인하려면 USER_TRIGGERS 데이터 사전을 조회한다.

- USER_TRIGGERS로 트리거 정보 조회하기

```
SELECT TRIGGER_NAME, TRIGGER_TYPE, TRIGGERING_EVENT, TABLE_NAME, STATUS
  FROM USER_TRIGGERS;
```

트리거 변경

- ALTER TRIGGER 명령어로 트리거 상태를 변경할 수 있다.
- 특정 트리거를 활성화 또는 비활성화하려면 ALTER TRIGGER 명령어에 ENABLE 또는 DISABLE 옵션을 지정한다.

```
ALTER TRIGGER 트리거 이름 ENABLE | DISABLE
```

- 특정 테이블과 관련된 모든 트리거의 상태를 활성화하거나 비활성화 하는 것도 가능하다.
  ⇒ 이 때 , ALTER TABLE 명령어를 사용한다.

```
특정 테이블과 관련된 모든 트리거의 상태 활성화
ALTER TABLE 테이블 이름 ENABLE ALL TRIGGERS;

특정 테이블과 관련된 모든 트리거의 상태 비활성화
ALTER TABLE 테이블 이름 DISABLE ALL TRIGGERS;
```

트리거 삭제

DROP 문을 사용하여 트리거를 삭제할 수 있다.

```
DROP TRIGGER 트리거 이름;
```

Q

1.

```
CREATE OR REPLACE PROCEDURE pro_dept_in
(
   inout_deptno IN OUT DEPT.DEPTNO%TYPE,
   out_dname OUT DEPT.DNAME%TYPE,
   out_loc OUT DEPT.LOC%TYPE
)
IS
BEGIN
   SELECT DEPTNO, DNAME, LOC INTO inout_deptno, out_dname, out_loc
     FROM DEPT
    WHERE DEPTNO = inout_deptno;
END pro_dept_in;
/
```

```
DECLARE
   v_deptno DEPT.DEPTNO%TYPE;
   v_dname DEPT.DNAME%TYPE;
   v_loc DEPT.LOC%TYPE;
BEGIN
   v_deptno := 10;
   pro_dept_in(v_deptno, v_dname, v_loc);
   DBMS_OUTPUT.PUT_LINE('부서번호 : ' || v_deptno);
   DBMS_OUTPUT.PUT_LINE('부서명 : ' || v_dname);
   DBMS_OUTPUT.PUT_LINE('지역 : ' || v_loc);
END;
/
```

2.

```
CREATE OR REPLACE FUNCTION func_date_kor(
   in_date IN DATE
)
RETURN VARCHAR2
IS
BEGIN
   RETURN (TO_CHAR(in_date, 'YYYY"년"MM"월"DD"일"'));
END func_date_kor;
/
```

3.

```
CREATE TABLE DEPT_TRG
    AS SELECT * FROM DEPT;
```

```
CREATE TABLE DEPT_TRG_LOG(
   TABLENAME   VARCHAR2(10), -- DML이 수행된 테이블 이름
   DML_TYPE    VARCHAR2(10), -- DML 명령어의 종류
   DEPTNO      NUMBER(2),    -- DML 대상이 된 부서번호
   USER_NAME   VARCHAR2(30), -- DML을 수행한 USER 이름
   CHANGE_DATE DATE          -- DML 이 수행된 날짜
);
```

```
CREATE OR REPLACE TRIGGER trg_dept_log
AFTER
INSERT OR UPDATE OR DELETE ON DEPT_TRG
FOR EACH ROW
BEGIN
   IF INSERTING THEN
     INSERT INTO DEPT_TRG_LOG
     VALUES ('DEPT_TRG', 'INSERT', :new.deptno,
             SYS_CONTEXT('USERENV', 'SESSION_USER'), sysdate);

   ELSIF UPDATING THEN
     INSERT INTO DEPT_TRG_LOG
     VALUES ('DEPT_TRG', 'UPDATE', :old.deptno,
             SYS_CONTEXT('USERENV', 'SESSION_USER'), sysdate);

   ELSIF DELETING THEN
     INSERT INTO DEPT_TRG_LOG
     VALUES ('DEPT_TRG', 'DELETE', :old.deptno,
             SYS_CONTEXT('USERENV', 'SESSION_USER'), sysdate);
   END IF;
END;
/
```