

## 21장 저장 프로시저, 저장 함수, 커서, 트리거

### 21.1 저장 프로시저

- 오라클은 사용자가 만든 PL/SQL 문을 데이터베이스에 저장 할 수 있도록 저장 프로시저 (stored procedure)라는 것을 제공한다.
- 이렇게 저장 프로시저를 사용하면 복잡한 DML 문들 필요할 때마다 다시 입력할 필요 없이 간단하게 호출만 해서 복잡한 DML 문의 실행 결과를 얻을 수 있다.
- 또한 저장 프로시저를 사용하면 성능도 향상되고, 호환성 문제도 해결된다.

#### 21.1.1 저장 프로시저 생성하는 방법

- 저장 프로시저를 생성하려면 CREATE PROCEDURE 다음에 새롭게 생성하고자하는 프로시저 이름을 기술한다.

```
-- 형식
CREATE [OR REPLACE ] PROCEDURE procedure_name (
    argument1 [mode] data_taye,
    argument2 [mode] data_taye, . . .)
IS
    local_variable declaration
BEGIN
    statement1;
    statement2;
    . . .
END;
/

-- 실행
EXECUTE procedure_name (argument1, argument2, ...);
```

#### [실습] 저장 프로시저 생성하기

- 사원 테이블에 저장된 모든 사원을 삭제하는 프로시저를 작성해 보도록 하겠다.

```
DROP TABLE EMP01;
CREATE TABLE EMP01
AS
SELECT * FROM EMP;

SELECT * FROM EMP01;

CREATE OR REPLACE PROCEDURE DEL_ALL
IS
BEGIN
    DELETE FROM EMP01;
END;
/

EXECUTE DEL_ALL;

SELECT * FROM EMP01;
```

### 21.1.2 저장 프로시저의 오류 원인 살피기

- 오류가 발생할 경우 "SHOW ERROR" 명령어를 수행하면 오류가 발생한 원인을 알 수 있게 된다.
- 원인을 분석하여 오류를 수정한 후 다시 저장 프로시저를 생성을 시도하여 '프로시저가 생성되었습니다.'란 메시지가 출력되어 저장 프로시저가 성공적으로 생성될 때까지 오류 수정 작업을 반복해야 한다.

```
DROP TABLE EMP01;
CREATE TABLE EMP01
AS
SELECT * FROM EMP;

SELECT * FROM EMP01;

DROP PROCEDURE DEL_ALL;
CREATE OR REPLACE PROCEDURE DEL_ALL
IS
--BEGIN
    DELETE FROM EMP01;
END;
/

SHOW ERROR
```

### 21.1.3 저장 프로시저 조회하기

- 저장 프로시저를 작성한 후 사용자가 저장 프로시저가 생성되었는지 확인하려면 USER\_SOURCE 살펴보면 된다.

```
DESC USER_SOURCE
SELECT NAME, TEXT FROM USER_SOURCE;
```

## 21.2 저장 프로시저의 매개 변수

- 저장 프로시저에 값을 전달해 주기 위해서 매개 변수를 사용한다.

```
DROP TABLE EMP01;
CREATE TABLE EMP01
AS
SELECT * FROM EMP;
SELECT * FROM EMP01;

CREATE OR REPLACE PROCEDURE
DEL_ENAME(ENAME EMP01.ENAME%TYPE)
IS
BEGIN
```

```

DELETE FROM EMP01 WHERE ENAME=VENAME;
END;
/

SELECT * FROM EMP01
WHERE ENAME='SMITH';

EXECUTE DEL_ENAME('SMITH');

SELECT * FROM EMP01
WHERE ENAME='SMITH';

```

## 21.3 IN, OUT, INOUT 매개 변수

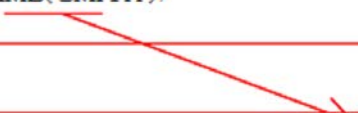
- CREATE PROCEDURE로 프로시저를 생성할 때 MODE를 지정하여 매개변수를 선언할 수 있는데 MODE 에 IN, OUT, INOUT 세 가지를 기술할 수 있다.
- IN 데이터를 전달 받을 때 쓰고 OUT은 수행된 결과를 받아갈 때 사용한다.
- INOUT은 두 가지 목적에 모두 사용된다.

### 21.3.1 IN 매개 변수

- 앞선 예제 중에서 매개변수로 사원의 이름을 전달받아서 해당 사원을 삭제하는 프로시저인 DEL\_ENAME를 작성해보았다.
- DEL\_ENAME 프로시저에서 사용된 매개변수는 프로시저를 호출할 때 기술한 값을 프로시저 내부에서 받아서 사용하고 있다.

```
EXECUTE DEL_ENAME('SMITH');
```

```
CREATE PROCEDURE DEL_ENAME(VENAME EMP01.ENAME%TYPE)
```



- 이렇게 프로시저 호출시 넘겨준 값을 받아오기 위한 매개변수는 MODE를 IN으로 지정해서 선언한다.

```
CREATE PROCEDURE DEL_ENAME(VENAME IN EMP01.ENAME%TYPE)
```

### 21.3.2 OUT 매개 변수

- 프로시저에 구한 결과 값을 얻어 내기 위해서는 MODE를 OUT으로 지정한다.

```

DROP PROCEDURE SEL_EMPNO;
CREATE OR REPLACE PROCEDURE SEL_EMPNO
( VEMPNO IN EMP.EMPNO%TYPE,
  VENAME OUT EMP.ENAME%TYPE,
  VSAL OUT EMP.SAL%TYPE,

```

```

VJOB OUT EMP.JOB%TYPE
)
IS

BEGIN

SELECT ENAME, SAL, JOB INTO VENAME, VSAL, VJOB
FROM EMP
WHERE EMPNO=VEMPNO;

END;
/

-- 바인드 변수
-- ':'를 덧붙여주는 변수는 미리 선언되어 있어야 한다.
VARIABLE VAR_ENAME VARCHAR2(15);
VARIABLE VAR_SAL NUMBER;
VARIABLE VAR_JOB VARCHAR2(9);

-- OUT 매개변수에서 값을 받아오기 위해서는 프로시저 호출 시 변수 앞에 ':'를 덧붙인다.
EXECUTE SEL_EMPNO(7788, :VAR_ENAME, :VAR_SAL, :VAR_JOB);

PRINT VAR_ENAME;

SELECT ENAME, SAL FROM EMP
WHERE EMPNO=7788;

```

## [과제] IN, OUT 매개변수 활용

- 사원명으로 검색하여 해당 사원의 직급을 얻어 오는 저장 프로시저를 SEL\_EMPNAME라는 이름으로 작성하라. (실습파일:CH21\_QUIZ\_01.SQL)

```

SQL> @CH21_QUIZ_01.SQL

프로시저가 생성되었습니다.

SQL> VARIABLE VAR_JOB VARCHAR2(9);
SQL> EXECUTE SEL_EMPNAME('SCOTT', :VAR_JOB);

PL/SQL 처리가 정상적으로 완료되었습니다.

SQL> PRINT VAR_JOB

VAR_JOB
-----
ANALYST

SQL>

```

<정답>

## 21.4 저장 함수

- 저장 함수(stored function)는 저장 프로시저와 거의 유사한 용도로 사용한다.
- 차이점이라곤 함수는 실행 결과를 되돌려 받을 수 있다는 점이다.

- 프로시저를 만들 때에는 PROCEDURE라고 기술하지만, 함수를 만들 때에는 FUNCTION이라고 기술한다.
- 함수는 결과를 되돌려 받기 위해서 함수가 되돌려 받게 되는 자료 형과 되돌려 받을 값을 기술해야 한다.
- 저장 함수는 호출결과를 얻어오기 위해서 호출 방식에 있어서도 저장 프로시저와 차이점이 있다.

```
-- 형식
create or replace function 함수이름(매개변수1 데이터타입, 매개변수2 데이터타입, ...)
return 데이터타입
is
    지역변수 데이터타입;
begin
    실행문;
    return 반환값;
end;
/

-- 실행
variable 바인드변수 데이터타입;
execute :바인드변수 := 함수이름(argument_list);
```

## [실습] 저장함수 작성하기

- 특별 보너스를 지급하기 위한 저장 함수를 작성해 보겠다. 보너스는 급여의 200%를 지급한다고 가정해 보겠다. (실습파일: PROC04.SQL)

```
SQL> CONN SCOTT/TIGER
SQL> ED PROC04

CREATE OR REPLACE FUNCTION CAL_BONUS(VEMPNO IN EMP.EMPNO%TYPE )
RETURN NUMBER
IS
    VSAL NUMBER(7, 2);
BEGIN
    SELECT SAL INTO VSAL
    FROM EMP
    WHERE EMPNO = VEMPNO;

    RETURN (VSAL * 2);
END;
/

SQL> @PROC04

함수가 생성되었습니다.

SQL> VARIABLE VAR_RES NUMBER;
SQL> EXECUTE :VAR_RES :=CAL_BONUS(7788);

PL/SQL 처리가 정상적으로 완료되었습니다.

SQL> PRINT VAR_RES
```

```

VAR_RES
-----
        6000

SQL> SELECT SAL, CAL_BONUS(7788)
2  FROM EMP
3  WHERE EMPNO=7788;

      SAL  CAL_BONUS(7788)
-----
      3000             6000

SQL>

```

### [실습] 판매된 도서에 대한 이익을 계산하는 함수

- 판매된 도서의 이익을 계산하기 위해 각 주문 건별로 실제 판매가격인 SALEPRICE를 입력받아 가격에 맞은 이익(30,000원 이상이면 10%, 30,000원 미만이면 5%)을 계산하여 반환하는 함수를 작성해 보자. (실습파일: FNC\_INTEREST.SQL)

```

D:\temp\sql>SQLPLUS MADANG/MADANG
SQL> ED FNC_INTEREST.SQL
CREATE OR REPLACE FUNCTION FNC_INTEREST(
  PRICE NUMBER) RETURN NUMBER
IS
  MYINTEREST NUMBER;
BEGIN
  -- 가격이 30,000원 이상이면 10%, 30,000원 미만이면 5%
  IF PRICE >= 30000 THEN MYINTEREST := PRICE * 0.1;
  ELSE MYINTEREST := PRICE * 0.05;
  END IF;
  RETURN MYINTEREST;
END;
/

SQL> @FNC_INTEREST.SQL

함수가 생성되었습니다.

SQL> SELECT CUSTID, ORDERID, SALEPRICE, FNC_INTEREST(SALEPRICE) INTEREST FROM ORDERS;

```

| CUSTID | ORDERID | SALEPRICE | INTEREST |
|--------|---------|-----------|----------|
| 1      | 1       | 6000      | 300      |
| 1      | 2       | 21000     | 1050     |
| 2      | 3       | 8000      | 400      |
| 3      | 4       | 6000      | 300      |
| 4      | 5       | 20000     | 1000     |
| 1      | 6       | 12000     | 600      |
| 4      | 7       | 13000     | 650      |
| 3      | 8       | 12000     | 600      |
| 2      | 9       | 7000      | 350      |
| 3      | 10      | 13000     | 650      |

```

10 개의 행이 선택되었습니다.

SQL>

```

## [과제] 저장 함수 생성

- 사원명으로 검색하여 해당 사원의 직급을 얻어 오는 저장 함수를 SEL\_EMPNAME02라는 이름으로 작성하라. (실습파일: CH21\_QUIZ\_02.SQL)

```
SQL> @CH21_QUIZ_02.SQL

프로시저가 생성되었습니다.

SQL> VARIABLE VAR_JOB VARCHAR2(9);
SQL> EXECUTE :VAR_JOB := SEL_EMPNAME02('SCOTT');

PL/SQL 처리가 정상적으로 완료되었습니다.

SQL> PRINT VAR_JOB

VAR_JOB
-----
ANALYST

SQL>
```

## 21.5 커서

- 대부분의 SELECT 문은 수행 후 반환되는 행의 개수가 한 개 이상이다.
- 처리 결과가 여러 개의 행으로 구해지는 SELECT 문을 처리하려면 커서를 이용해야 한다.
- 커서(cursor)는 실행 결과 테이블을 한 번에 한 행씩 처리하기 위하여 테이블의 행을 순서대로 가리키는 데 사용한다.

### [표] 커서와 관련된 키워드

| 키워드                           | 역할         |
|-------------------------------|------------|
| CURSOR <cursor 이름> IS <커서 정의> | 커서를 생성     |
| OPEN <cursor 이름>              | 커서의 사용을 시작 |
| FETCH <cursor 이름> INTO <변수>   | 행 데이터를 가져옴 |
| CLOSE <cursor 이름>             | 커서의 사용을 끝냄 |

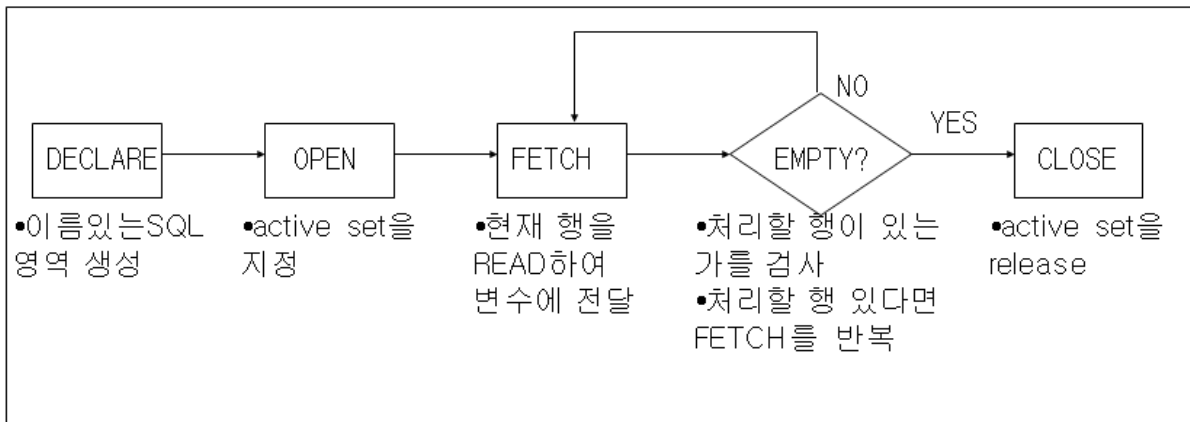
### 21.5.1 커서의 형식과 개념

```
-- 형식
DECLARE
    CURSOR cursor_name IS statement; -- 커서 선언
BEGIN
    OPEN cursor_name; -- 커서 열기
```

```

FETCH cur_name INTO variable_name; --커서로부터 데이터를 읽어와 변수에 저장
CLOSE cursor_name; --커서 닫기
END;

```



### (1) DECLARE CURSOR

- 명시적으로 CURSOR를 선언하기 위해 CURSOR문장을 사용한다.

```

-- 형식
CURSOR cursor_name IS
select_statement;

-- 예
CURSOR C1 IS
SELECT * FROM DEPT;

```

### (2) OPEN CURSOR

- 질의를 수행하고 검색 조건을 충족하는 모든 행으로 구성된 결과 셋을 생성하기 위해 CURSOR를 OPEN한다. CURSOR는 이제 결과 셋에서 첫 번째 행을 가리킨다.

```

-- 형식
OPEN cursor_name;

OPEN c1;

```

### (3) FETCH CURSOR

- FETCH 문은 결과 셋에서 로우 단위로 데이터를 읽어 들인다. 각 인출(FETCH) 후에 CURSOR는 결과 셋에서 다음 행으로 이동한다.

```

-- 형식
FETCH cursor_name INTO {variable1[,variable2, . . . .]};

-- 예시

```



```

LOOP
    FETCH C1 INTO VDEPT.DEPTNO, VDEPT.DNAME, VDEPT.LOC;
    EXIT WHEN C1%NOTFOUND;
END LOOP;

```

#### (4) 커서의 상태

- FETCH 문을 설명하면서 커서의 속성 중에 NOTFOUND를 언급하였는데 오라클에서는 이외에도 다양한 커서의 속성을 통해 커서의 상태를 알려주는데 이 속성을 이용해서 커서를 제어해야 한다.

| 속성        | 의미                              |
|-----------|---------------------------------|
| %NOTFOUND | 커서 영역의 자료가 모두 FETCH됐었다면 TRUE    |
| %FOUND    | 커서 영역에 FETCH 되지 않은 자료가 있다면 TRUE |
| %ISOPEN   | 커서가 OPEN된 상태이면 TRUE             |
| %ROWCOUNT | 커서가 얻어 온 레코드의 개수                |

#### (5) CLOSE CURSOR

- CLOSE문장은 CURSOR를 사용할 수 없게 하고 결과 셋의 정의를 해제한다.
- SELECT 문장이 다 처리된 완성 후에는 CURSOR를 닫는다.

```

-- 형식
CLOSE cursor_name;

```

#### [실습] 부서 테이블의 모든 내용을 조회하기

- 커서를 사용하여 부서 테이블의 모든 내용을 출력한다.

1. ED 다음에 파일이름을 입력하여 새로 생긴 SQL 파일에 다음과 같이 입력한다. (실습파일: PROC05.SQL)

```

[PROC05.SQL]

01  SET SERVEROUTPUT ON
02  CREATE OR REPLACE PROCEDURE CURSOR_SAMPLE01
03  IS
04      VDEPT DEPT%ROWTYPE;
05      CURSOR C1 -- 커서 선언
06      IS
07      SELECT * FROM DEPT;
08  BEGIN
09      DBMS_OUTPUT.PUT_LINE('부서번호 / 부서명 / 지역명');
10      DBMS_OUTPUT.PUT_LINE('-----');
11
12      OPEN C1; -- 커서 열기
13
14      LOOP
15          FETCH C1 INTO VDEPT.DEPTNO, VDEPT.DNAME, VDEPT.LOC; -- 커서로부터 데이터를 읽어와 변수에 저장
16
17          EXIT WHEN C1%NOTFOUND;

```

```

18
19         DBMS_OUTPUT.PUT_LINE(VDEPT.DEPTNO||
20         ' '||VDEPT.DNAME||' '||VDEPT.LOC);
21     END LOOP;
22
23     CLOSE C1; -- 커서 닫기
24 END;
25 /

```

2. 작성을 완료한 후에 파일을 저장한다. SQL> 프롬프트에서 @파일명을 입력하면 SQL 파일 내부에 기술한 PL/SQL이 실행된 후 오류가 발생하지 않으면 '프로시저가 생성되었습니다'라는 메시지와 함께 프로시저가 생성된다. 다음과 같이 프로시저를 호출하면 부서 테이블의 모든 정보가 출력된다.

```

SQL> @PROC05

프로시저가 생성되었습니다.

SQL> EXECUTE CURSOR_SAMPLE01;
부서번호 / 부서명 / 지역명
-----
10 ACCOUNTING NEW YORK
20 RESEARCH DALLAS
30 SALES CHICAGO
40 OPERATIONS BOSTON

PL/SQL 처리가 정상적으로 완료되었습니다.

SQL>

```

## 21.5.2 CURSOR와 FOR LOOP

- CURSOR FOR LOOP는 명시적 CURSOR에서 행을 처리합니다. LOOP에서 각 반복마다 CURSOR를 열고 행을 인출(FETCH)하고 모든 행이 처리되면 자동으로 CURSOR가 CLOSE되므로 사용하기가 편리합니다.

```

-- 형식
FOR record_name IN cursor_name LOOP
statement1;
statement2;
. . . . .
END LOOP

```

### [실습] 부서 테이블의 모든 내용을 조회하기 (CURSOR와 FOR LOOP)

- OPEN ~ FETCH ~ CLOSE가 없이 FOR ~ LOOP ~ END LOOP문을 사용하여 보다 간단하게 커서를 처리할 수 있다. 커서를 사용하여 부서 테이블의 모든 내용을 출력해 보겠다.

1. ED 다음에 파일이름을 입력하여 새로 생긴 SQL 파일에 다음과 같이 입력한다. (실습파일: PROC06.SQL)

```
[PROC06.sql]
```

```

01 SET SERVEROUTPUT ON
02 CREATE OR REPLACE PROCEDURE CURSOR_SAMPLE02
03 IS
04     VDEPT DEPT%ROWTYPE;
05     CURSOR C1
06     IS
07     SELECT * FROM DEPT;
08 BEGIN
09     DBMS_OUTPUT.PUT_LINE('부서번호 / 부서명 / 지역명');
10     DBMS_OUTPUT.PUT_LINE('-----');
11     FOR VDEPT IN C1 LOOP
12         EXIT WHEN C1%NOTFOUND;
13         DBMS_OUTPUT.PUT_LINE(VDEPT.DEPTNO||
14             ' '||VDEPT.DNAME||' '||VDEPT.LOC);
15     END LOOP;
16 END;
17 /

```

2. 작성을 완료한 후에 파일을 저장한다. SQL> 프롬프트에서 @파일명을 입력하면 SQL 파일 내부에 기술한 PL/SQL이 실행된 후 오류가 발생하지 않으면 '프로시저가 생성되었습니다'라는 메시지와 함께 프로시저가 생성된다. 다음과 같이 프로시저를 호출하면 부서 테이블의 모든 정보가 출력된다.

```

SQL> @PROC06

프로시저가 생성되었습니다.

SQL> EXECUTE CURSOR_SAMPLE02;
부서번호 / 부서명 / 지역명
-----
10 ACCOUNTING NEW YORK
20 RESEARCH DALLAS
30 SALES CHICAGO
40 OPERATIONS BOSTON

PL/SQL 처리가 정상적으로 완료되었습니다.

SQL>

```

## 21.6 트리거

### 21.6.1 트리거란 무엇인가?

- 해당 단어의 의미처럼 어떤 이벤트가 발생하면 자동적으로 방아쇠가 당겨져 총알이 발사되듯이 특정 테이블이 변경되면 이를 이벤트로 다른 테이블이 자동으로 변경되도록 하기 위해서 사용한다.
- 트리거는 특정 동작을 이벤트로 인해서만 실행되는 프로시저의 일종이다.
- 트리거를 만들기 위한 CREATE TRIGGER의 형식은 다음과 같다.

```

CREATE TRIGGER trigger_name
timing[BEFORE|AFTER] event[INSERT|UPDATE|DELETE]
ON table_name
[FOR EACH ROW]

```

```
[WHEN conditions]
BEGIN
    statement
END
```

#### ■ 트리거의 타이밍

- [BEFORE] 타이밍은 어떤 테이블에 INSERT, UPDATE, DELETE 문이 실행될 때 해당 문장이 실행되기 전에 트리거가 가지고 있는 BEGIN ~ END 사이의 문장을 실행한다.
- [AFTER] 타이밍은 INSERT, UPDATE, DELETE 문이 실행되고 난 후에 트리거가 가지고 있는 BEGIN ~ END 사이의 문장을 실행한다.

#### ■ 트리거의 이벤트

- 사용자가 어떤 DML(INSERT, UPDATE, DELETE)문을 실행했을 때 트리거를 발생시킬 것인지를 결정한다.

#### ■ 트리거의 몸체

- 해당 타이밍에 해당 이벤트가 발생하게 되면 실행될 기본 로직이 포함되는 부분으로 BEGIN ~ END에 기술한다.

#### ■ 트리거의 유형

- 트리거의 유형은 FOR EACH ROW에 의해 문장 레벨 트리거와 행 레벨 트리거로 나눈다.
- FOR EACH ROW가 생략되면 문장 레벨 트리거이고 행 레벨 트리거를 정의하고자 할 때에는 반드시 FOR EACH ROW를 기술해야만 한다.
- 문장 레벨 트리거는 어떤 사용자가 트리거가 설정되어 있는 테이블에 대해 DML(INSERT, UPDATE, DELETE)문을 실행할 때 단 한번만 트리거를 발생시킬 때 사용한다.
- 행 레벨 트리거는 DML(INSERT, UPDATE, DELETE)문에 의해서 여러 개의 행이 변경된다면 각 행이 변경될 때마다 트리거를 발생시키는 방법이다. 만약 5개의 행이 변경되면 5번 트리거가 발생된다.

#### ■ 트리거의 조건

- 트리거 조건은 행 레벨 트리거에서만 설정할 수 있으며 트리거 이벤트에 정의된 테이블에 이벤트가 발생할 때 보다 구체적인 데이터 검색 조건을 부여할 때 사용된다.

### [실습] 단순 메시지를 출력하는 트리거 작성하기

- 사원 테이블에 새로운 데이터가 들어오면 '신입사원이 입사했습니다.'란 메시지를 출력하도록 문장 레벨 트리거로 작성한다.

```
-- 1. 사원 테이블 생성
DROP TABLE EMP01 CASCADE CONSTRAINTS;
CREATE TABLE EMP01(
    EMPNO NUMBER(4) PRIMARY KEY,
    ENAME VARCHAR2(20),
    JOB VARCHAR2(20)
);

-- 2. 트리거 작성(TRIG01.SQL)
CREATE OR REPLACE TRIGGER TRG_01
AFTER INSERT
ON EMP01
BEGIN
    DBMS_OUTPUT.PUT_LINE('신입사원이 입사했습니다.');
```

```
END;
```

```

/

-- 3. 사원 테이블에 로우를 추가한다.
SET SERVEROUTPUT ON
INSERT INTO EMP01 VALUES(1, '전원지', '화가');

```

## [실습] 급여 정보를 자동으로 추가하는 트리거 작성하기

- 사원 테이블에 새로운 데이터가 들어오면(즉, 신입 사원이 들어오면) 급여 테이블에 새로운 데이터(즉 신입 사원의 급여 정보)를 자동으로 생성하도록 하기 위해서 사원 테이블에 트리거를 작성한다. 즉, 신입사원의 급여는 일괄적으로 100으로 한다.

```

--1. 급여를 저장할 테이블을 생성한다.
DROP TABLE SAL01;
CREATE TABLE SAL01(
SALNO NUMBER(4) PRIMARY KEY,
SAL NUMBER(7,2),
EMPNO NUMBER(4) REFERENCES EMP01(EMPNO)
);

--2. 급여번호를 자동 생성하는 시퀀스를 정의하고 이 시퀀스로부터 일련번호를 얻어 급여번호에 부여한다.
CREATE SEQUENCE SAL01_SALNO_SEQ;

--3. 트리거 생성한다.
CREATE OR REPLACE TRIGGER TRG_02
AFTER INSERT
ON EMP01
FOR EACH ROW
BEGIN
INSERT INTO SAL01 VALUES(
SAL01_SALNO_SEQ.NEXTVAL, 100, :NEW.EMPNO);
END;
/

--4. 사원 테이블에 로우를 추가합니다
INSERT INTO EMP01 VALUES(2, '전수빈', '프로그래머');
SELECT * FROM EMP01;
SELECT * FROM SAL01;

INSERT INTO EMP01 VALUES(3, '김종현', '교수');
SELECT * FROM EMP01;
SELECT * FROM SAL01;

```

## [실습] 급여 정보를 자동으로 삭제하는 트리거 작성하기

- 사원이 삭제되면 그 사원의 급여 정보도 자동 삭제되는 트리거를 작성한다.

```

-- 1. 사원 테이블의 로우를 삭제한다.
DELETE FROM EMP01 WHERE EMPNO=2; -- SQL Error 발생한다.

-- 2. 트리거를 작성한다.
CREATE OR REPLACE TRIGGER TRG_03
AFTER DELETE ON EMP01
FOR EACH ROW
BEGIN
DELETE FROM SAL01 WHERE EMPNO=:old.EMPNO;

```

```
END;
/

-- 3. 사원 테이블의 로우를 삭제한다.
DELETE FROM EMP01 WHERE EMPNO=2;
SELECT * FROM EMP01;
SELECT * FROM SAL01;
```

## 21.6.2 트리거 삭제

- DROP TIGGER 다음에 삭제할 트리거 명을 기술합니다.

```
DROP TRIGGER TRG_02;
```

### [실습] 급여 정보를 자동 추가하는 트리거 제거하기

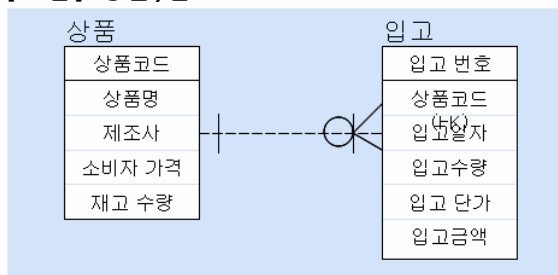
1. 급여 정보를 자동 추가하는 트리거인 TRG\_02를 제거해 본다.
2. TRG\_02 트리거를 삭제하고 난 후에 사원 테이블에 행을 추가하였더니 급여가 자동적으로 입력되지 않는 것을 확인 할 수 있다.

```
DROP TRIGGER TRG_02;
INSERT INTO EMP01 VALUES(4, '최은정', '선생님');
SELECT * FROM EMP01;
SELECT * FROM SAL01;
```

## 21.6.3 예제를 통한 트리거의 적용

- 상품 테이블의 예제를 통해서 실질적인 트리거의 적용 예를 살펴보도록 한다.

### [그림] 상품,입고 ERD



### [실습] 입고 트리거 작성하기

- 입고 테이블에 상품이 입력되면 입고 수량을 상품 테이블의 재고 수량에 추가하는 트리거 작성한다.

```
-- 1. 테이블을 생성한다.
CREATE TABLE 상품(
```

```

상품코드 CHAR(6) PRIMARY KEY,
상품명 VARCHAR2(12) NOT NULL,
제조사 VARCHAR(12),
소비자가격 NUMBER(8),
재고수량 NUMBER DEFAULT 0
);

CREATE TABLE 입고(
입고번호 NUMBER(6) PRIMARY KEY,
상품코드 CHAR(6) REFERENCES 상품(상품코드),
입고일자 DATE DEFAULT SYSDATE,
입고수량 NUMBER(6),
입고단가 NUMBER(8),
입고금액 NUMBER(8)
);

-- 2. 샘플 데이터를 입력한다.
INSERT INTO 상품(상품코드, 상품명, 제조사, 소비자가격)
VALUES('A00001', '세탁기', 'LG', 500);
INSERT INTO 상품(상품코드, 상품명, 제조사, 소비자가격)
VALUES('A00002', '컴퓨터', 'LG', 700);
INSERT INTO 상품(상품코드, 상품명, 제조사, 소비자가격)
VALUES('A00003', '냉장고', '삼성', 600);
SELECT * FROM 상품;

-- 3. 입고 트리거
CREATE OR REPLACE TRIGGER TRG_04
AFTER INSERT ON 입고
FOR EACH ROW
BEGIN
UPDATE 상품
SET 재고수량 = 재고수량 + :NEW.입고수량
WHERE 상품코드 = :NEW.상품코드;
END;
/

-- 4. 입고 테이블에 상품을 입력한다.
INSERT INTO 입고(입고번호, 상품코드, 입고수량, 입고단가, 입고금액)
VALUES(1, 'A00001', 5, 320, 1600);
SELECT * FROM 입고;
SELECT * FROM 상품;

INSERT INTO 입고(입고번호, 상품코드, 입고수량, 입고단가, 입고금액)
VALUES(2, 'A00002', 10, 680, 6800);
SELECT * FROM 입고;
SELECT * FROM 상품;

INSERT INTO 입고(입고번호, 상품코드, 입고수량, 입고단가, 입고금액)
VALUES(3, 'A00003', 3, 220, 660);
INSERT INTO 입고(입고번호, 상품코드, 입고수량, 입고단가, 입고금액)
VALUES(4, 'A00003', 5, 220, 1100);
SELECT * FROM 입고;
SELECT * FROM 상품;

```

## [실습] 갱신 트리거 작성하기

- 이미 입고된 상품에 대해서 입고 수량이 변경되면 상품 테이블의 재고수량 역시 변경되어야 한다. 이를 위한 갱신 트리거를 작성한다.

```

-- 1. 갱신 트리거 (파일이름: TRG05.SQL)
CREATE OR REPLACE TRIGGER TRG03

```

```

AFTER UPDATE ON 입고
FOR EACH ROW
BEGIN
UPDATE 상품
SET 재고수량 = 재고수량 + (-:old.입고수량+:new.입고수량)
WHERE 상품코드 = :new.상품코드;
END;
/

-- 2. 입고 수량 변경
UPDATE 입고 SET 입고수량=10, 입고금액=2200
WHERE 입고번호=3;
SELECT * FROM 입고 ORDER BY 입고번호;
SELECT * FROM 상품;

```

## [실습] 삭제 트리거 작성하기

- 입고 테이블에서 입고되었던 상황이 삭제되면 상품 테이블에 재고수량에서 삭제된 입고수량만큼을 빼는 삭제 트리거 작성한다.

```

-- 1. 삭제트리거
CREATE OR REPLACE TRIGGER TRG04
AFTER DELETE ON 입고
FOR EACH ROW
BEGIN
UPDATE 상품
SET 재고수량 = 재고수량 - :old.입고수량
WHERE 상품코드 = :old.상품코드;
END;
/

-- 2. 삭제
DELETE 입고 WHERE 입고번호=3;
SELECT * FROM 입고 ORDER BY 입고번호;
SELECT * FROM 상품;

```

## [표] 프로시저, 트리거, 사용자 정의 함수의 공통점과 차이점

|        | 저장 프로시저                      | 트리거   | 저장 함수                          |
|--------|------------------------------|---|--------------------------------|
| 공통점    | 오라클의 경우 PL/SQL로 작성           |   |                                |
| 정의 방법  | CREATE PROCEDURE 문           | CREATE TRIGGER 문                                    | CREATE FUNCTION 문              |
| 호출 방법  | EXEC 문으로 직접 호출               | INSERT, DELETE, UPDATE<br>문이 실행될 때 자동으로 실행됨         | SELECT 문으로 호출                  |
| 기능의 차이 | SQL 문으로 할 수 없는 복잡한<br>로직을 수행 | 기본 값 제공, 데이터 제약 준수,<br>SQL 뷰의 수정, 참조무결성 작업<br>등을 수행 | 속성 값을 가공하여 반환,<br>SQL 문에 직접 사용 |



## [과제] 과제-21-01.TXT

- SQL> CONN ORA\_USER/HONG 로 접속하여 SQL문을 작성한다.
- 부서번호, 부서명, 작업 flag(I: insert, U:update, D:delete)을 매개변수로 받아 CH10\_DEPARTMENTS 테이블에 각각 INSERT, UPDATE, DELETE 하는 CH10\_IUD\_DEP\_PROC 란 이름의 프로시저를 작성하라. 다음과 같이 부서테이블의 복사본을 만든다.

```
--1. 환경설정
CREATE TABLE CH10_DEPARTMENTS
AS
SELECT DEPARTMENT_ID, DEPARTMENT_NAME
FROM DEPARTMENTS;

ALTER TABLE CH10_DEPARTMENTS ADD CONSTRAINTS PK_CH10_DEPARTMENTS PRIMARY KEY (DEPARTMENT_ID);

--2. 실행내용
EXECUTE CH10_IUD_DEP_PROC(1,'ACCOUNTING','I');
EXECUTE CH10_IUD_DEP_PROC(1,'SALES','U');
EXECUTE CH10_IUD_DEP_PROC(1,'SALES','D');

SELECT * FROM CH10_DEPARTMENTS
ORDER BY DEPARTMENT_ID;
```

<정답>

## [과제] 과제-21-02.TXT

- SQL> CONN SCOTT/TIGER 로 접속하여 SQL문을 작성한다.
- 부서 번호를 전달하여 해당 부서 소속 사원의 정보를 출력하는 SEL\_EMP 프로시저를 커서를 사용하여 작성하라.

```
SQL> EXECUTE SEL_EMP(20);

사원번호 / 사원명 / 직급 / 급여
-----
7369 SMITH CLERK 800
7566 JONES MANAGER 2975
7788 SCOTT ANALYST 3000
```

<정답>