절차적 프로그래밍

Stored Procedure

- Stored Procedure
  - ✓ 관계형 데이터베이스에서도 절차적 프로그래밍이 가능
  - ✓ 기본 형식
    - 선언

```
DELIMITER $$
    CREATE PROCEDURE 프로시저이름()
    BEGIN
      SQL 프로그래밍
    END $$
DELIMITER ;
```

    - 호출

```
CALL 프로시저이름();
```

      - ◆ DELIMITER $$ ~ END $$ 부분까지 프로시저의 코딩할 부분을 묶어줌
      - ◆ MariaDB의 종료 문자는 세미콜론(;)인데 CREATE PROCEDURE 안에서도 세미콜론이 종료 문자이므로 어디까지가 프로시저인지 구별이 어렵기 때문에 END $$가 나올 때까지를 프로시저로 인식하게 함
      - ◆ DELIMITER로 종료 문자를 세미콜론(;) 으로 변경해 놓아야 하는데 CALL 프로시저이름();은 CREATE PROCEDURE로 생성한 프로시저를 호출

Stored Procedure

- Stored Procedure
  - ✓ 사용 목적
    - 성능 향상
    - 유지 및 관리가 간편
    - 모듈화 프로그래밍 가능
    - 보안 강화

Stored Procedure

- IF ~ ELSE
  - ✓ 조건에 따라 분기
  - ✓ 한 문장 이상이 처리되어야 할 때는 BEGIN... END와 함께 묶어줘야 함
  - ✓ 기본 형식

```
IF <부울 표현식> THEN
        SQL문장들
ELSE
        SQL문장들
END IF;
```

Stored Procedure

- ❖ IF ~ ELSE
  - ✓ 입사한지 5년이 지났는지 확인하는 프로시저
    - ● 프로시저 생성

```
DELIMITER //
CREATE PROCEDURE ifProc()
BEGIN
        DECLARE hireDATE DATE; -- 입사일
        DECLARE curDATE DATE; -- 오늘
        DECLARE days INT; -- 근무한 일수

        SELECT HIREDATE INTO hireDate --HIREDATE 열의 값을 hireDATE에 대입
          FROM EMP
          WHERE EMPNO = 7369;

        SET curDATE = CURRENT_DATE(); -- 현재 날짜
        SET days =  DATEDIFF(curDATE, hireDATE); -- 날짜의 차이, 일 단위

        IF (days/365) >= 5 THEN -- 5년이 지났다면
                SELECT CONCAT('입사한지 5년이 지났습니다.');
        ELSE
                SELECT '입사한지 아직 5년이 되지 않았습니다.' ;
        END IF;
END //
DELIMITER ;
```

Stored Procedure

- IF ~ ELSE
  - ✓ 입사한지 5년이 지났는지 확인하는 프로시저
    - 프로시저 호출

      CALL ifProc();

Stored Procedure

- ❖ IF ~ ELSE
  - ✓ 점수를 이용한 학점 계산
    - ● 프로시저 생성

```
DELIMITER //
CREATE PROCEDURE ifElseProc()
BEGIN
    DECLARE point INT ;
    DECLARE credit CHAR(1);
    SET point = 77 ;

    IF point >= 90 THEN
                    SET credit = 'A';
    ELSEIF point >= 80 THEN
                    SET credit = 'B';
    ELSEIF point >= 70 THEN
                    SET credit = 'C';
    ELSEIF point >= 60 THEN
                    SET credit = 'D';
    ELSE
                    SET credit = 'F';
    END IF;
    SELECT CONCAT('취득점수==>', point), CONCAT('학점==>', credit);
END //
DELIMITER ;
```

Stored Procedure

- IF ~ ELSE
  - ✓ 점수를 이용한 학점 계산
    - 프로시저 호출

```
CALL ifElseProc();
```

Stored Procedure

- CASE ~ WHEN
  - ✓ 분기
  - ✓ 기본 형식

```
CASE
        WHEN <부울 표현식> THEN
                SQL문장들
        ELSE
                SQL문장들
END CASE;
```

Stored Procedure

- CASE ~ WHEN
  - ✓ 점수를 이용한 학점 계산
    - ● 프로시저 생성

```
DELIMITER //
CREATE PROCEDURE caseProc()
BEGIN
    DECLARE point INT ;
    DECLARE credit CHAR(1);
    SET point = 77 ;
    CASE
                WHEN point >= 90 THEN
                        SET credit = 'A';
                WHEN point >= 80 THEN
                        SET credit = 'B';
                WHEN point >= 70 THEN
                        SET credit = 'C';
                WHEN point >= 60 THEN
                        SET credit = 'D';
                ELSE
                        SET credit = 'F';
    END CASE;
    SELECT CONCAT('취득점수==>', point), CONCAT('학점==>', credit);
END //
DELIMITER ;
```

Stored Procedure

- CASE ~ WHEN
  - ✓ 점수를 이용한 학점 계산
    - 프로시저 호출

```
CALL caseProc();
```

Stored Procedure

- WHILE
  - ✓ 반복문
  - ✓ 기본 형식

```
WHILE <부울 식> DO
        SQL 명령문들...
END WHILE;
```

Stored Procedure

- WHILE
  - ✓ 반복문을 이용한 피보나치 수열
    - 프로시저 생성

```
DELIMITER //
CREATE PROCEDURE fibonacci()
BEGIN
    DECLARE i INT; -- 3에서 10까지 증가할 변수
    DECLARE f1 INT;
    DECLARE f2 INT;
    DECLARE hap INT; -- 더한 값을 누적할 변수
   SET i = 3;
    SET f1 = 1;
   SET f2 = 1;
   SET hap = f1 + f2;
    WHILE (i <= 10) DO
       SET hap = f1 + f2;
       SET f1 = f2;
       SET f2 = hap;
       SET i = i + 1;
    END WHILE;
    SELECT hap;
END //
DELIMITER ;
```

Stored Procedure

- WHILE
  - ✓ 반복문을 이용한 피보나치 수열
    - 프로시저 호출

      CALL fibonacci();

Stored Procedure

- WHILE
  - ✓ ITERATE 는 지정한 레이블로 이동하는 명령어
  - ✓ LEAVE 는 반복문을 종료하기 위한 명령어

Stored Procedure

- WHILE
  - ✓ ITERATE 와 LEAVE
    - 프로시저 생성

```
CREATE PROCEDURE whileProc()
BEGIN
    DECLARE i INT; -- 1에서 100까지 증가할 변수
    DECLARE hap INT; -- 더한 값을 누적할 변수
    SET i = 1;
    SET hap = 0;
    myWhile: WHILE (i <= 100) DO  -- While문에 label을 지정
     IF (i%7 = 0) THEN
        SET i = i + 1;
        ITERATE myWhile; -- 지정한 label문으로 가서 계속 진행
     END IF;
        SET hap = hap + i;
        IF (hap > 1000) THEN
        LEAVE myWhile; -- 지정한 label문을 떠남. 즉, While 종료.
     END IF;
        SET i = i + 1;
    END WHILE;
    SELECT hap;
END //
DELIMITER ;
```

Stored Procedure

- WHILE
  - ✓ ITERATE 와 LEAVE
    - 프로시저 호출

      CALL whileProc();

Stored Procedure

- ❖ 오류 처리
  - ✓ 프로시저 실행 도중 에러가 발생하는 경우에 대한 처리를 제공
  - ✓ 기본 형식

    DECLARE 액션

    HANDLER FOR 오류 조건 처리할_문장;

    - 액션: 오류 발생 시 행동을 정의하는 것으로 CONTINUE 와 EXIT 둘 중 하나를 사용하는데 CONTINUE가 나오면 처리할 문장을 수행하고 EXIT를 설정하면 종료
    - 오류 조건: 어떤 오류를 처리할 것 인지를 지정하는 것으로 MariaDB의 오류 코드 숫자가 오거나 SQLSTATE '상태코드' 또는 SQLEXCEPTION, SQLWARNING, NOT FOUND 등이 올 수 있는데 SQLSTATE에서 상태 코드는 5자리 문자열로 되어 있고 SQLEXCEPTION은 예외를 SQLWARNING은 경고 메시지가 출력되는 경우 NOT FOUND는 커서나 SELECT ~ INTO에서 발생되는 오류를 의미
    - 처리할_문장: 처리할 문장이 여러 개일 경우에는 BEGIN ~ END로 묶어줌
    - MariaDB의 오류 코드 1000 ~ 1885, 1900 ~ 1981 까지 정의되어 있는데 https://mariadb.com/kb/en/mariadb-error-codes/ 에서 확인 가능

Stored Procedure

- ❖ 오류 처리
  - ✓ 테이블이 존재하지 않는 경우 메시지 출력
    - 프로시저 생성

```
DELIMITER //
CREATE PROCEDURE errorProc()
BEGIN
    DECLARE CONTINUE HANDLER FOR 1146 SELECT '테이블이 존재하지 않습니다.' AS '메시지';
    SELECT * FROM EMPS;  -- noTable은 없음.
END //
DELIMITER ;
```

    - 프로시저 호출

```
CALL errorProc();
```

Stored Procedure

- ❖ 오류 처리
  - ✓ 테이블이 존재하지 않는 경우 메시지 출력
    - ● 프로시저 생성

```
DELIMITER //
CREATE PROCEDURE insertError()
BEGIN
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    BEGIN
                SHOW ERRORS; -- 오류 메시지 출력
                SELECT '오류가 발생했네요. 작업은 취소시켰습니다.' AS '메시지';
                ROLLBACK; -- 오류 발생시 작업을 롤백
    END;
    INSERT INTO userTBL VALUES('adam', '박문석', 1970, '제주', NULL,
                NULL, 170, CURRENT_DATE());
END //
DELIMITER ;
```

    - ● 프로시저 호출

```
CALL insertError();
```

Stored Procedure

- 동적 SQL
  - ✓ SQL을 미리 준비해 두었다가 필요할 때 수행하고 해제하는 것
  - ✓ PREPARE 로 SQL을 준비하고 EXECUTE로 실행하고 DEALLOCATE PREFARE 로 해제
  - ✓ SQL을 작성할 때 ?로 매개변수를 받을 수 있도록 만들고 USING으로 매개변수를 받아서 실행하는 것도 가능
  - ✓ 테이블의 데이터를 전부 조회하는 동적 SQL
    - PREPARE selectQuery FROM 'SELECT * FROM EMP';
    - EXECUTE selectQuery;
    - DEALLOCATE PREPARE selectQuery;

Stored Procedure

- 동적 SQL
  - ✓ 매개변수를 받아서 테이블의 데이터를 조회하는 동적 SQL
    - PREPARE paramQuery FROM 'SELECT * FROM EMP WHERE empno = ?';
    - SET @empno = 7788;
    - EXECUTE paramQuery USING @empno;
    - DEALLOCATE PREPARE paramQuery;

Stored Procedure

- 매개변수 활용
  - ✓ 매개변수가 있는 프로시저 생성

```
DELIMITER $$
CREATE PROCEDURE 스토어드 프로시저이름( IN 또는 OUT 파라미터 ) BEGIN
        SQL 프로그래밍
END $$
DELIMITER ;
```

  - ✓ 매개변수가 있는 프로시저 생성

```
CALL 스토어드 프로시저이름(매개변수 나열);
```

Stored Procedure

❖ 매개변수 활용

✓ 매개변수가 있는 프로시저

● 프로시저 생성

```
DELIMITER //
create PROCEDURE myproc(vuserid char(15), vname varchar(20)CHARACTER set utf8, vbirthyear int(11),
vaddr char(100)CHARACTER set utf8, vmobile char(11), vmdate date)
begin
        INSERT INTO usertbl
        VALUES(vuserid, vname, vbirthyear, vaddr, vmobile, vmdate);
end//
DELIMITER ;
```

● 프로시저 실행

```
call myproc('BoA', '권보아', 1986, '남양주', '01012341234', '1986-11-5');
```

● 확인

```
select * from usertbl;
```

Stored Procedure

- 프로시저 삭제

```
drop procedure myproc;
```

TRIGGER

- TRIGGER
  - ✓ 테이블에 사건이 발생했을 때 데이터베이스에서 자동적으로 절차적 프로그래밍 블록을 수행시키기 위해서 사용하는 개체
  - ✓ TRIGGER는 TRIGGER링 이벤트가 일어날 때마다 암시적으로 실행
  - ✓ TRIGGER링 이벤트는 데이터베이스 테이블에 수행되는 INSERT, UPDATE, DELETE 오퍼레이션
  - ✓ TRIGGER가 사용되는 경우
    - 테이블 생성시 CONSTRAINT로 선언 제한이 불가능하고 복잡한 무결성 제한을 유지
    - DML문장을 사용한 사람,변경한 내용,시간 등을 기록함으로써 정보를 AUDIT하기
    - 테이블을 변경할 때 일어나야 할 동작을 다른 테이블 또는 다른 프로그램 들에게 자동적으로 신호하기
  - ✓ TRIGGER에 대한 제한
    - TRIGGER는 트랜잭션 제어 문장(COMMIT,ROLLBACK,SAVEPOINT)을 사용하지 못함
    - TRIGGER 주요부에 의해 호출되는 프로시저나 함수는 트랜잭션 제어 문장을 사용하지 못함
    - TRIGGER 주요부는 LONG또는 LONG RAW변수를 선언할 수 없음
    - TRIGGER 주요부가 액세스하게 될 테이블에 대한 제한이 있음

TRIGGER

❖ TRIGGER를 만들기 위한 CREATE TRIGGER 문의 형식

CREATE TRIGGER *TRIGGER_name*

timing[BEFORE|AFTER] *event*[INSERT|UPDATE|DELETE]

ON *table_name*

[FOR EACH ROW]

[WHEN conditions]

BEGIN

statement

END

- ✓ TRIGGER_name: TRIGGER의 식별자
- ✓ BEFORE | AFTER: DML문장이 실행되기 전에 TRIGGER를 실행할 것인지 실행된 후에 TRIGGER를 실행할 것인지를 정의
- ✓ TRIGGERing_event: TRIGGER를 실행하는 DML(INSERT,UPDATE,DELETE)문을 기술
- ✓ OF column: TRIGGER가 실행되는 테이블에서 COLUMN명을 기술
- ✓ table_name: TRIGGER가 실행되는 테이블 이름
- ✓ FOR EACH ROW: 이 옵션을 사용하면 행 레벨 TRIGGER가 되어 TRIGGERing문장에 의해 영향받은 행에 대해 각각 한번씩 실행하고 사용하지 않으면 문장 레벨 TRIGGER가 되어 DML문장 당 한번만 실행

TRIGGER

- TRIGGER를 만들기 위한 CREATE TRIGGER 문의 형식
  - ✓ TRIGGER의 타이밍
    - [BEFORE] 타이밍은 어떤 테이블에 INSERT, UPDATE, DELETE 문이 실행될 때 해당 문장이 실행되기 전에 TRIGGER가 가지고 있는 BEGIN ~ END 사이의 문장을 실행
    - [AFTER] 타이밍은 INSERT, UPDATE, DELETE 문이 실행되고 난 후에 TRIGGER가 가지고 있는 BEGIN ~ END 사이의 문장을 실행
  - ✓ TRIGGER의 이벤트
    - 사용자가 어떤 DML(INSERT, UPDATE, DELETE)문을 실행했을 때 TRIGGER를 발생시킬 것인지를 결정
  - ✓ TRIGGER의 몸체
    - 해당 타이밍에 해당 이벤트가 발생하게 되면 실행될 기본 로직이 포함되는 부분으로 BEGIN ~ END에 기술
    - IF INSERTING THEN 문장을 이용해서 삽입, 갱신(UPDATING), 삭제(DELETING) 시 다른 작업 수행 가능

TRIGGER

- TRIGGER를 만들기 위한 CREATE TRIGGER 문의 형식
  - ✓ TRIGGER의 유형
    - TRIGGER의 유형은 FOR EACH ROW에 의해 문장 레벨 TRIGGER와 행 레벨 TRIGGER로 나눔
    - FOR EACH ROW가 생략되면 문장 레벨 TRIGGER이고 행 레벨 TRIGGER를 정의하고자 할 때에는 반드시 FOR EACH ROW를 기술
    - 문장 레벨 TRIGGER는 어떤 사용자가 TRIGGER가 설정되어 있는 테이블에 대해 DML(INSERT, UPDATE, DELETE)문을 실행할 때 단 한번만 TRIGGER를 발생시킬 때 사용
    - 행 레벨 TRIGGER는 DML(INSERT, UPDATE, DELETE)문에 의해서 여러 개의 행이 변경된다면 각 행이 변경될 때마다 TRIGGER를 발생시키는 방법입니다. 만약 5개의 행이 변경되면 5번 TRIGGER가 발생
  - ✓ TRIGGER에서 OLD와 NEW
    - 행 레벨 TRIGGER에서만 사용할 수 있는 예약어로 TRIGGER 내에서 현재 처리되고 있는 행을 액세스할 수 있는데 두개의 의사 레코드를 통하여 이 작업을 수행할 수 있는데 OLD는 INSERT문에 의해 정의되지 않고 NEW는 DELETE에 대해 정의되지 않지만 UPDATE는 OLD와 NEW를 모두 정의
    - INSERT모든 필드는 NULL로 정의 – 문장이 완전할 때 삽입된 새로운 값
    - UPDATE 갱신하기 전의 원래 값 – 문장이 완전할 때 갱신된 새로운 값
    - DELETE 행이 삭제되기 전의 원래 값 – 모든 필드는 NULL

TRIGGER

- TRIGGER 활용
  - ✓ 삽입 트리거

```
CREATE TABLE EMP01(
    EMPNO INT PRIMARY KEY,
    ENAME VARCHAR(10) NOT NULL,
    JOB VARCHAR(20)
);

CREATE TABLE SAL01(
    SALNO INT PRIMARY KEY AUTO_INCREMENT,
    SAL FLOAT(7,2),
    EMPNO INT REFERENCES EMP01(EMPNO) ON DELETE CASCADE
);
```

TRIGGER

- TRIGGER 활용
  - ✓ 삽입 트리거

```
DELIMITER //
CREATE TRIGGER TRG_01
AFTER INSERT
ON EMP01
FOR EACH ROW
BEGIN
INSERT INTO SAL01(SAL, EMPNO) VALUES(100, NEW.EMPNO);
END //
DELIMITER ;

INSERT INTO EMP01
VALUES(2, '박문석', '프로그래머');


SELECT * FROM EMP01;

SELECT * FROM SAL01;
```

TRIGGER

- TRIGGER 활용
  - ✓ 삭제 트리거

```
DELIMITER //
CREATE TRIGGER TRG_02
AFTER DELETE ON EMP01
FOR EACH ROW
BEGIN
DELETE FROM SAL01 WHERE EMPNO=OLD.EMPNO;
END //
DELIMITER ;

DELETE FROM EMP01 WHERE EMPNO=1;

SELECT * FROM EMP01;

SELECT * FROM SAL01;
```

TRIGGER

❖ 트리거 삭제 – DROP TRIGGER 트리거이름

```
DROP TRIGGER TRG_01;

DROP TRIGGER TRG_02;
```

TRIGGER

- TRIGGER 활용
  - ✓ EMP 테이블에서 급여를 수정 시 현재의 값보다 적게 수정할 수 없으며 현재의 값보다 10% 이상 높게 수정할 수 없도록 하는 TRIGGER를 작성

```
DELIMITER //
CREATE TRIGGER emp_sal_chk
BEFORE UPDATE ON EMP
FOR EACH ROW
BEGIN
    IF (NEW.sal < OLD.sal OR NEW.sal > OLD.sal * 1.1) THEN
        SIGNAL SQLSTATE '50000'
       SET MESSAGE_TEXT = '잘못된 수정입니다..';
   END IF;
END //
DELIMITER ;

UPDATE EMP
SET SAL = SAL * 1.2;

SELECT * FROM EMP;
```

TRIGGER

- TRIGGER 활용
  - ✓ 다중 트리거 – 하나의 테이블에 여러 개의 트리거를 생성하는 것
  - ✓ 중첩 트리거 – 트리거가 다른 트리거를 호출하는 구조

TRIGGER

❖ TRIGGER 활용

```
CREATE TABLE IF NOT EXISTS testTbl (id INT, txt VARCHAR(10));
INSERT INTO testTbl VALUES(1, '소녀시대');
INSERT INTO testTbl VALUES(2, '트와이스');
INSERT INTO testTbl VALUES(3, '블랙핑크');

DELIMITER //
CREATE TRIGGER testTrg  -- 트리거 이름
    AFTER  DELETE - 삭제 후에 작동하도록 지정
    ON testTbl -- 트리거를 부착할 테이블
    FOR EACH ROW -- 각 행마다 적용시킴
BEGIN
        SET @msg = '가수 그룹이 삭제됨' ; -- 트리거 실행시 작동되는 코드들
END //
DELIMITER ;
```

TRIGGER

- TRIGGER

```
SET @msg = '';

INSERT INTO testTbl VALUES(4, '에스파');
SELECT @msg;

UPDATE testTbl SET txt = 'ITZY' WHERE id = 2;
SELECT @msg;

DELETE FROM testTbl WHERE id = 4;
SELECT @msg;
```