데이터베이스

Chapter #05
데이터베이스 프로그래밍

컴퓨터공학과
박지용

무단 복제 및 배포를 금지합니다.

데이터베이스 프로그래밍

데이터베이스 프로그래밍의 개념

❖ 프로그래밍이란?

- 프로그램을 설계하고 소스코드를 작성하여 디버깅하는 과정

❖ 데이터베이스 프로그래밍이란?

- DBMS에 데이터를 정의하고 저장된 데이터를 읽어와 데이터를 변경하는 프로그램을 작성하는 과정
- 일반 프로그래밍과는 데이터베이스 언어인 SQL을 포함한다는 점이 다름

(a) SQL Interface

(b) 삽입 프로그래밍

그림 5-1 데이터베이스 프로그래밍

데이터베이스 프로그래밍의 개념

❖ 데이터베이스 프로그래밍 방법

❶ SQL 전용 언어를 사용하는 방법

SQL 자체의 기능을 확장하여 변수, 제어, 입출력 등의 기능을 추가한 새로운 언어를 사용하는 방법.

오라클은은 스토어드 프로시저 언어를 사용하며, SQL Server는 T-SQL이라는 언어를 사용함.

❷ 일반 프로그래밍 언어에 SQL을 삽입하여 사용하는 방법

자바, C, C++ 등 일반 프로그래밍 언어에 SQL 삽입하여 사용하는 방법.

일반 프로그래밍 언어로 작성된 응용 프로그램에서 데이터베이스에 저장된 데이터를 관리, 검색함.

삽입된 SQL문은 DBMS의 컴파일러가 처리함.

❸ 웹 프로그래밍 언어에 SQL을 삽입하여 사용하는 방법

호스트 언어가 JSP, ASP, PHP 등 웹 스크립트 언어인 경우다.

❹ 4GL(4th Generation Language)

데이터베이스 관리 기능과 비주얼 프로그래밍 기능을 갖춘 'GUI 기반 소프트웨어 개발 도구'를 사용하여 프로그래밍하는 방법. Delphi, Power Builder, Visual Basic 등이 있음.

데이터베이스 프로그래밍의 개념

단계적으로 database management system과 프로그램이 구성이 된다.

그림 5-2 **DBMS 플랫폼과 데이터베이스 프로그래밍의 유형**

데이터베이스 프로그래밍의 개념

사실상 쌍둥이 비슷.
MySQL + α = Oracle
└ 개업에 지방만 기능

표 5-1 DBMS의 종류와 특징

특징	Access	SQL Server	Oracle	MySQL	DB2	SQLite
제조사	마이크로 소프트사	마이크로 소프트사	오라클사	오라클사	IBM사	리처드 힙 (오픈소스)
운영체제 기반	윈도우	윈도우 리눅스	윈도우, 유닉스, 리눅스	윈도우, 유닉스, 리눅스	유닉스	모바일 OS (안드로이드, iOS 등)
용도	개인용 DBMS	윈도우 기반 기업용 DBMS	대용량 데이터 베이스를 위한 응용	~~소용량~~ 대용량 데이터 베이스를 위한 응용	대용량 데이터 베이스를 위한 응용	모바일 전용 데이터베이스

└ 우리가 여기 사용

대부분 리눅스를 지원함.

└ 내장 기반제품으로 사용되는 db=SQLite

MySQL 샘플 데이터 설치

❖ 실습 데이터베이스 설치

- **sqldb.sql 다운로드**
  - https://github.com/jwpark-sungshin/sqldb/blob/master/sqldb.sql

MySQL 샘플 데이터 설치

❖ 실습 데이터베이스 설치

- **sqldb.sql 다운로드**
  - https://github.com/jwpark-sungshin/sqldb/blob/master/sqldb.sql

MySQL 샘플 데이터 설치

❖ 실습 데이터베이스 설치

- **sqldb.sql 다운로드**
  - https://github.com/jwpark-sungshin/sqldb/blob/master/sqldb.sql
    - 파일 이름에서 txt 확장자 삭제
    - 저장 타입은 All Files로 변경

MySQL 샘플 데이터 설치

❖ sqldb 데이터베이스 생성

```
show databases;
create database sqldb;
use sqldb;
```

스토어드 프로시저와 함수

쿼리 여러개 항상 같이 실행된다면
하나로 묶어서 연속적으로 한 번에 처리
→ 자주 모듈화 (항상 연속적으로 호출하는) 쿼리들 모듈화해서 모듈 형태로 스토어드 프로시저 만들어 놓음.

스토어드 프로시저 → MySQL에서 사용할 수 있는 일종의 프로그래밍 기능.

스토어드 프로시저의 개요

- 스토어드 프로시저(Stored Procedure, 저장 프로시저)
  - MySQL에서 제공되는 프로그래밍 기능
  - 쿼리문의 집합으로 어떠한 동작을 일괄 처리하기 위한 용도로 사용
  - 쿼리 모듈화
    - 필요할 때마다 호출만 하면 훨씬 편리하게 MySQL 운영 ← 여러 쿼리문.
    - CALL 프로시저_이름( ) 으로 호출 → 한번에 해당 쿼리 쭉 실행할 수 있음.

스토어드 프로시저

스토어드 프로시저의 개요

- 기본 형식

형식 :

종결문자에 혼동하고 부분에도 ; 들어감 → 원래 SQL은 ; 만나면 해당 명령 끝났다고 생각.
여기를 세팅을 해줘서 임시로 끝나는 거를 특정 변수자로 바꿔줌.
└ delimiter

```
DELIMITER $$
CREATE PROCEDURE 스토어드 프로시저이름( IN 또는 OUT 파라미터 )
BEGIN

    이 부분에 SQL 프로그래밍 코딩..
```

└ 여기에 프로시저 코딩.

```
END $$
```

끝나면 임시로 만들어줬던 delimiter 사용.

```
DELIMITER ;
```

delimiter 원래대로 바꿔줌.

실제로 사용할 때 →

```
CALL 스토어드 프로시저이름();
```

실제 프로그래밍 언어에서 호출하듯이
인자 있으면 인자값 넣어줌.
인자 없으면 빈 괄호

스토어드 프로시저

스토어드 프로시저의 개요

- 스토어드 프로시저 생성 예

```
1  USE sqlDB;
2  DROP PROCEDURE IF EXISTS userProc;
3  DELIMITER $$
4  CREATE PROCEDURE userProc()
5  BEGIN
6      SELECT * FROM userTbl; -- 스토어드 프로시저 내용
7  END $$
8  DELIMITER ;
9
10 CALL userProc();
```

안쪽에 있던 프로시저 삭제

없어도 됨. 항상 userProc 이라는 프로시저 drop
존재하지 않으면 에러

이제 프로시저를 만들거임!

앞에서 봤듯이 스토어드 프로시저 만듦.

select문 대체

이렇게 호출해주면 우리가 마지 line 6 select문 실행한 것처럼 똑같은 결과가 나옴.

스토어드 프로시저

스토어드 프로시저의 개요

- 스토어드 프로시저의 수정과 삭제
  - 수정 : ALTER PROCEDURE
  - 삭제 : DROP PROCEDURE
- 매개 변수의 사용

함수이름 정의할 때 IN/OUT 있고 없음

  - 입력 매개 변수를 지정하는 형식

```
IN 입력_매개변수_이름 데이터_형식
```

INT
VCHAR(10)

input

(out → output)

  - 입력 매개 변수가 있는 스토어드 프로시저 실행 방법

```
CALL 프로시저_이름(전달 값);
```

호출시, ( ) 안에 매개변수 넣어줌

스토어드 프로시저

스토어드 프로시저의 개요

- 매개 변수의 사용
  - 출력 매개 변수 지정 방법

```
OUT 출력_매개변수_이름 데이터_형식
```

output

  - 출력 매개 변수에 값 대입하기 위해 주로 SELECT... INTO문 사용
  - 출력 매개 변수가 있는 스토어드 프로시저 실행 방법

```
CALL 프로시저_이름(@변수명);
SELECT @변수명;
```

매개변수를 임시로 만들 때.

@

@a → a라는 변수를 만들 때

이렇게 임시변수 만들고

여기다가 나중에 스토어드 프로시저 안에서 계산한 결과 저장 → select ... into 로 할 때

매개변수 이름

변수에 결과 받아서 그 결과값을 select문 통해 확인.

스토어드 프로시저

스토어드 프로시저의 개요

- 스토어드 프로시저 실습
  - 입력 매개 변수 1개의 스토어드 프로시저 생성

```
USE sqlDB;
DROP PROCEDURE IF EXISTS userProc1;
DELIMITER $$
CREATE PROCEDURE userProc1(IN userName VARCHAR(10))
BEGIN
  SELECT * FROM userTbl WHERE name = userName;
END $$
DELIMITER ;

CALL userProc1('조관우');
```

input으로 들어온 userName 여기다 사용

varchar형이어야만 함.

	userID	name	birthYear	addr	mobile1	mobile2	height	mDate
▶	JKW	조관우	1965	경기	018	9999999	172	2010-10-10

스토어드 프로시저

스토어드 프로시저의 개요

- 스토어드 프로시저 실습
  - 입력 매개 변수 2개의 스토어드 프로시저 생성

```
DROP PROCEDURE IF EXISTS userProc2;
DELIMITER $$
CREATE PROCEDURE userProc2(
    IN userBirth INT,
    IN userHeight INT
)
BEGIN
  SELECT * FROM userTbl
    WHERE birthYear > userBirth AND height > userHeight;
END $$
DELIMITER ;

CALL userProc2(1970, 178);
```

조건으로 input값을 이렇게 넣기

T8-2 9:05

나이가 더 많고 키가 더 작은 사람을 select (반대인듯)

	userID	name	birthYear	addr	mobile1	mobile2	height	mDate
▶	LSG	이승기	1987	서울	011	1111111	182	2008-08-08
	SSK	성시경	1979	서울	NULL	NULL	186	2013-12-12

스토어드 프로시저

스토어드 프로시저의 개요

- 스토어드 프로시저 실습
  - 출력 매개변수 설정
    - 스토어드 프로시저 생성 및 테스트로 사용할 테이블 생성

```
DROP PROCEDURE IF EXISTS userProc3;
DELIMITER $$
CREATE PROCEDURE userProc3(
    IN txtValue CHAR(10),
    OUT outValue INT
)
BEGIN
  INSERT INTO testTBL VALUES(NULL,txtValue);
  SELECT MAX(id) INTO outValue FROM testTBL;
END $$
DELIMITER ;
```

output을 이 변수를 통해서 내보낼 것임

input

→ 가장 큰 id 찾아서 outValue에 넣음

스토어드 프로시저

스토어드 프로시저의 개요

- 스토어드 프로시저 실습
  - 출력 매개변수 설정
    - 스토어드 프로시저 생성 및 테스트로 사용할 테이블 생성

```
CREATE TABLE IF NOT EXISTS testTBL(
    id INT AUTO_INCREMENT PRIMARY KEY,
    txt CHAR(10)
);
```

자동으로 id가 하나씩 늘어남.

내부적으로 max(id) 값 바꿈

auto_increment.
가장 마지막에 insert한 값+1이 결국 가장 큰 id값. 가정.

input

```
CALL userProc3 ('테스트값', @myValue);
SELECT CONCAT('현재 입력된 ID 값 ==>', @myValue);
```

→ 테스트값이라는 내용이 testTBL에 insert됨. (자동으로 id increment)

→ 넣을 때마다 0,1,2,3 이렇게 늘어남

→ 늘어난 값은 myValue에 넣어줌.

2개의 문자열을 합침

출력 결과:

현재 입력된 ID 값 ==> 1

처음 하나 입력시

스토어드 프로시저

스토어드 프로시저의 개요

- 스토어드 프로시저 실습
  - 스토어드 프로시저 안에 SQL 프로그래밍 활용
    - While문 활용한 2중 반복문

100글자까지 한글도 있는 가변 대사형

```
DROP TABLE IF EXISTS guguTBL;
CREATE TABLE guguTBL (txt VARCHAR(100)); -- 구구단 저장용 테이블

DROP PROCEDURE IF EXISTS whileProc;
DELIMITER $$
CREATE PROCEDURE whileProc()
BEGIN
    DECLARE str VARCHAR(100); -- 각 단을 문자열로 저장
    DECLARE i INT; -- 구구단 앞자리
    DECLARE k INT; -- 구구단 뒷자리
    SET i = 2; -- 2단부터 계산

```

└이 컬럼 하나만 가정

스토어드 프로시저

스토어드 프로시저의 개요

- 스토어드 프로시저 실습
  - 스토어드 프로시저 안에 SQL 프로그래밍 활용
    - While문 활용한 2중 반복문

조건

```
    WHILE (i < 10) DO -- 바깥 반복문. 2단~9단까지
        SET str = ''; -- 각 단의 결과를 저장할 문자열 초기화
        SET k = 1; -- 구구단 뒷자리는 항상 1부터 9까지
        WHILE (k < 10) DO
            SET str = CONCAT(str, '  ', i, 'x', k, '=', i*k); -- 문자열 만들기
            SET k = k + 1; -- 뒷자리 증가
        END WHILE;
        SET i = i + 1; -- 앞자리 증가
        INSERT INTO guguTBL VALUES(str); -- 각 단의 결과를 테이블에 입력
    END WHILE;
END $$
DELIMITER ;

CALL whileProc();
SELECT * FROM guguTBL;
```

결과들을 다 저장할 문자열

이전 결과에 계속 덧붙여 줌.

구구단 앞자리 구구단 뒷자리

출력되는 것 X. 구구단 결과가 guguTBL에 insert 되어있는 것.

스토어드 프로시저

스토어드 프로시저의 개요

- 스토어드 프로시저 실습
  - 스토어드 프로시저 안에 SQL 프로그래밍 활용
    - While문 활용한 2중 반복문

9-2 18:32

하나의 row에 이거 다 들어있음.

txt
2x1=2 2x2=4 2x3=6 2x4=8 2x5=10 2x6=12 2x7=14 2x8=16 2x9=18
3x1=3 3x2=6 3x3=9 3x4=12 3x5=15 3x6=18 3x7=21 3x8=24 3x9=27
4x1=4 4x2=8 4x3=12 4x4=16 4x5=20 4x6=24 4x7=28 4x8=32 4x9=36
5x1=5 5x2=10 5x3=15 5x4=20 5x5=25 5x6=30 5x7=35 5x8=40 5x9=45
6x1=6 6x2=12 6x3=18 6x4=24 6x5=30 6x6=36 6x7=42 6x8=48 6x9=54
7x1=7 7x2=14 7x3=21 7x4=28 7x5=35 7x6=42 7x7=49 7x8=56 7x9=63
8x1=8 8x2=16 8x3=24 8x4=32 8x5=40 8x6=48 8x7=56 8x8=64 8x9=72
9x1=9 9x2=18 9x3=27 9x4=36 9x5=45 9x6=54 9x7=63 9x8=72 9x9=81

스토어드 프로시저

스토어드 프로시저의 개요

- 스토어드 프로시저 실습
  - 현재 저장된 프로시저의 이름 및 내용 확인
  - INFORMATION_SCHEMA 데이터베이스의 ROUTINES 테이블을 조회하면 내용 확인 가능
    - SELECT routine_name, routine_definition FROM INFORMATION_SCHEMA.ROUTINES WHERE routine_schema = 'sqldb' AND routine_type = 'PROCEDURE';

우리가 사용한 db의 이름

실제 어떻게 프로시저가 만이 구현되어 있는지..
begin에서 end까지

이거 모르겠고 적으면 답 안 나옴...

ROUTINE_NAME	ROUTINE_DEFINITION
caseProc	BEGIN DECLARE bYear INT; DECLARE tti CHAR(3); -- 띠 SELECT birthYear into bYear FROM userTbl WHERE name = userName; CASE WHEN ...
errorProc	BEGIN DECLARE i INT; -- 1씩 증가하는 값 DECLARE hap INT UNSIGNED; -- 합계 DECLARE saveHap INT UNSIGNED; -- 합계 DECLARE EXIT ...
ifelseProc	BEGIN DECLARE bYear INT; -- 변수 선언 SELECT birthYear into bYear FROM userTbl WHERE name = userName; IF (bYear >= 1980) THEN SELECT '...
userProc	BEGIN SELECT * FROM userTbl; -- 스토어드 프로시저 내용 END
userProc1	BEGIN SELECT * FROM userTbl WHERE name = userName; END
userProc2	BEGIN SELECT * FROM userTbl WHERE birthYear > userBirth AND height > userHeight; END
userProc3	BEGIN INSERT INTO testTBL VALUES(NULL,txtValue); SELECT MAX(id) INTO outValue FROM testTBL; END
whileProc	BEGIN DECLARE str VARCHAR(100); -- 각 단을 문자열로 저장 DECLARE i INT; -- 구구단 앞자리 DECLARE k INT; -- 구구단 뒷자리 SET i = 2; -...

스토어드 프로시저

스토어드 프로시저의 개요

- 스토어드 프로시저 실습
  - 파라미터를 확인하려면 INFORMATION_SCHEMA 데이터베이스의 ~~ROUTINES~~ PARAMETERS 테이블 조회
    - SELECT parameter_mode, parameter_name, dtd_identifier
      FROM INFORMATION_SCHEMA.PARAMETERS
      WHERE specific_name = 'userProc3';
      ㄴ에 대한 파라미터값을 가져온 것

	PARAMETER_MODE	PARAMETER_NAME	DTD_IDENTIFIER
▶	IN	txtValue	char(10)
	OUT	outValue	int(11)

스토어드 프로시저

스토어드 프로시저의 개요

- 스토어드 프로시저 실습
  - SHOW CREATE PROCEDURE문으로도 스토어드 프로시저의 내용 확인 가능
    - SHOW CREATE PROCEDURE sqldb.userProc3;

스토어드 프로시저

스토어드 프로시저의 특징

간단히 편의상으로만 좋아지는게 아니라

- MySQL의 성능 향상 → 보통 MySQL을 원격에서 사용(외부서버 사용) → 네트워크를 통해서 쿼리가 전달 → 정상적으로 쿼리 수행하려면 그 쿼리의 내용이 그대로 MySQL 서버로 전달이 되어야함.
  → 프로시저 만들면서 그냥 call 프로시저이름(); → 프로시저에 대한 내용 이미 MySQL서버에 저장되어 있음.
  → 쿼리에 대한 내용은 전송할 필요X. 필요한 인자들만 전송이 되면 됨.
  이것만 client→server로 전송되면 됨.
  이런 프로시저 사용한건가 이런 인자 넘겨줄건지
  network상 전송되는 양 줄일 수 있음.
  - 긴 쿼리가 아니라 짧은 프로시저 내용만 클라이언트에서 서버로 전송
    - 네트워크 부하 줄임으로 MySQL의 성능 향상

- 유지관리가 간편
  - 응용 프로그램에서는 프로시저만 호출
    - 데이터베이스에서 관련된 스토어드 프로시저의 내용 수정/유지보수

정상 이능 바꿀수있지

응용프로그램 상에서 SQL 쿼리 안들어서 끼기에 편함. 문제는 없지만 결국 응용프로그램과 데이터베이스는 분리가 되어있음
db상에서 db 관리자가 table 변경을 하면 응용프로그램 상에서는 쿼리가 동작을 하지 않을 수 있음
→ db 관리자와 응용프로그램 관리자가 따로 있기 때문에 그런 것
이런 과정을 다 db쪽으로 넘기면 응용프로그램은 그냥 프로시저만 콜함.

응용프로그램쪽에서는 db가 어떻게 변하는지 신경쓸 필요 없이 해당 프로시저만 콜을 하는 방식.

스토어드 프로시저

스토어드 프로시저의 특징

- 모듈식 프로그래밍 가능

  모듈식 프로그래밍 → 함수 단위로 쪼개고 쪼개서 그냥 부르면 됨.
  하나를 잘 만들기 놓으면 그거를 여러 다른 함수에서 불러다 사용할 수도 있음.

  - 언제든지 실행 가능 call 프로시저이름();

    프로시저 안에서 다른 프로시저 호출 가능.
  - 스토어드 프로시저로 저장해 놓은 쿼리의 수정, 삭제 등의 관리 수월
  - 모듈식 프로그래밍 언어와 동일한 장점 갖음

- 보안 강화

  이건지 아닌 정보 안 주기 위해서 뷰 사용. 권한 설정

  테이블에 민감한 정보 / 프로시저 활용해 좀 더 효과적으로 사용자한테 권한을 적게 줄 수 있음.

  - 사용자 별로 테이블 접근 권한 주지 않고 스토어드 프로시저에만 접근 권한을 주어 보안 강화

    내가 의도한대로만 동작하기끔 강제 가능

    - 뷰 또한 스토어드 프로시저와 같이 보안 강화 가능

    사용자가 실제 데이터 보지 않고도 프로시저 호출해 결과값만 받을 수 있게 만들어 줌. (약국 내가 만들어놓은 대로만 동작)

    테이블에 대한 접근권 주면 그 테이블에 어떤 쿼리 날릴지 예측 불가. 악용가능.

스토어드 함수 ≠ 스토어드 프로시저

스토어드 함수 (Stored Function)

- 사용자가 직접 만들어서 사용하는 함수

mysql의 내장함수 (max, min ...)
→ select문 앞에 붙여서 사용됨. select max(...) ...
뒤

- 스토어드 프로시저와 유사 사용 방식이 다름.
  - 형태와 사용 용도에 있어 차이 있음 (프로시저랑은 사용할 수 있는 위치가 다름)
- 스토어드 함수의 개요

```
DELIMITER $$
CREATE FUNCITON 스토어드 함수이름( 파라미터 )
    RETURNS 반환형식
BEGIN

    이 부분에 프로그래밍 코딩..
    RETURN 반환값;

END $$
DELIMITER ;
SELECT 스토어드_함수이름();
```

procedure 아님

항상 반환하는 부분이 있어야... → select문 뒤에 붙기 때문.

INT

실제 사용시) → 반환값은 출력되어야... select문이랑 붙여서 사용

스토어드 함수

스토어드 함수와 스토어드 프로시저의 차이점

- 스토어드 함수
  - 파라미터에 IN, OUT 등을 사용할 수 없음
    - 모두 입력 파라미터로 사용
  - 아나까 이런 거는 따로 인자로 받지 X.
  - **RETURNS문**으로 반환할 값의 데이터 형식 지정 (프로시저는 out 선언시 값 이름, data 형식 같이 지정)
    - (out의 역할)
    - **본문 안에서는 RETURN문**으로 하나의 값 반환
  - SELECT 문장 안에서만 호출
  - 스토어드
  - 함수 안에서 집합 결과 반환하는 SELECT 사용 불가
    - 결과가 여러개 나오는 것들.
    - SELECT... INTO... 는 집합 결과 반환하는 것이 아니므로 예외적으로 스토어드 함수에서 사용 가능
    - 출력하는 게 X
  - 어떤 계산 통해서 하나의 값 반환하는데 주로 사용
    - 스토어드 함수 〃 하나의 값 반환

스토어드 함수

스토어드 함수와 스토어드 프로시저의 차이점

좀 더 다양한 사용 가능

- 스토어드 프로시저
  - 파라미터에 IN, OUT 등을 사용 가능
  - 별도의 반환하는 구문이 없음 ~~returns~~
    - 필요하다면 여러 개의 OUT 파라미터 사용해서 값 반환 가능

      select into 활용해 어떤 변수에 값을 집어넣는 걸로 반환 대신.
  - CALL로 호출
  - 스토어드 프로시저 안에 SELECT문 사용 가능 (자유롭게)
  - 여러 SQL문이나 숫자 계산 등의 다양한 용도로 사용

스토어드 함수

스토어드 함수와 스토어드 프로시저의 차이점

- 스토어드 함수
  - 스토어드 함수를 사용하기 위해서는 스토어드 함수 생성 권한을 허용 해야함

우리가 사용하고 있는 mysql 8.0 version의 경우.

```
SET GLOBAL log_bin_trust_function_creators = 1;
```

영구적으로 세팅 → mysql의 config 파일 수정
그냥 이거는 임시로 세팅(mysql 서버가 꺼지거나 하면 다시 초기화)

  - ex) 2개의 숫자의 합계를 계산하는 스토어드 함수

```
USE sqlDB;
DROP FUNCTION IF EXISTS userFunc;
DELIMITER $$
CREATE FUNCTION userFunc(value1 INT, value2 INT)
    RETURNS INT
BEGIN
    RETURN value1 + value2;
END $$
DELIMITER ;

SELECT userFunc(100, 200);
```

스토어드 함수

스토어드 함수 실습(P. 455~ 456)

- 저장해 놓았던 sqlDB.sql 이용해 sqlDB 데이터베이스 초기화
- 출생년도 입력하면 나이 출력되는 함수 작성
  - 테이블을 조회할 때 주로 사용되는 함수

```
USE sqlDB;
DROP FUNCTION IF EXISTS getAgeFunc;
DELIMITER $$
CREATE FUNCTION getAgeFunc(bYear INT)
    RETURNS INT
BEGIN
    DECLARE age INT;
    SET age = YEAR(CURDATE()) - bYear;
    RETURN age;
END $$
DELIMITER ;
```

input

나이

리턴 타입

mysql에서 제공하는 함수

오늘이 해당하는 날짜와 년도.

날짜와 년도에서 년도만 따로 뺌.

만 나이

스토어드 함수

스토어드 함수 실습(P. 455~ 456)

나이는 db에 입력 후 매년 값이 바뀜. → 매년 해당 컬럼을 직접 업데이트.
└이거저 알고 필요할 때마다 나이 계산하자 (함수이용)

- 함수 호출
  - SELECT getAgeFunc(1979);
  - 1979년생의 현재 나이가 출력됨

미리 변수에

- 함수의 반환값을 SELECT … INTO … 로 저장했다가 사용 가능

아예 input자를 줄 수도

```
SELECT getAgeFunc(1979) INTO @age1979;
SELECT getAgeFunc(1997) INTO @age1989;
SELECT CONCAT('1997년과 1979년의 나이차 ==> ', (@age1979-@age1989));
```

  - 두 출생년도의 나이차가 출력됨

스토어드 함수

스토어드 함수 실습(P. 455~ 456)

- 함수는 주로 테이블을 조회할 때 활용됨
  - SELECT userID, name, getAgeFunc(birthYear) AS '만 나이' FROM userTbl;

	userID	name	만 나이
▶	BBK	바비킴	46
	EJW	은지원	47
	JKW	조관우	54
	JYP	조용필	69
	KBS	김범수	40
	KKH	김경호	48
	LJB	임재범	56
	LSG	이승기	32
	SSK	성시경	40
	YJS	윤종신	50

스토어드 함수

스토어드 함수 실습(P. 455~ 456)

- 현재 저장된 스토어드 함수의 이름 및 내용 확인
  - SHOW CREATE FUNCTION getAgeFunc;

- 스토어드 함수 삭제
  - 다른 데이터베이스 개체와 마찬가지로 DROP문 사용
  - DROP FUNCTION getAgeFunc;

트리거

트리거

트리거(Trigger)의 개요

어떤 data가 지워질 때 항상 다른 table의 어떤 작업을 연결해서 실행을 해야 되는 경우가 있음. → 이런거 자동화 가능.

ex) 쇼핑몰에서 어떤 사람이 구매 → 우리에 재고에서 마이너스하고 그거를 배송에 넘겨주기..

(이런 과정을 트리거 형식으로 만들 수 있음)

- 트리거란?
  - 사전적 의미로 '방아쇠'
  - 방아쇠 당기면 '자동'으로 총알이 나가듯이 테이블에 무슨 일이 일어나면 '자동'으로 실행
  - 제약 조건과 더불어 데이터 무결성을 위해 MySQL에서 사용할 수 있는 기능

    table 안 data를 조작할 때만 event 발생해서 트리거 작동
  - 테이블에 DML문(Insert, Update, Delete 등) 이벤트가 발생될 때 작동
  - 테이블에 부착되는 프로그램 코드
  - 직접 실행 불가 (스토어드 프로시저처럼 우리가 언제든지 호출할 수 있는게 X)
    - 테이블에 이벤트 일어나야 자동 실행
  - IN, OUT 매개 변수를 사용할 수 없음
  - MySQL은 View에 트리거 부착 불가

    임시로 만든 가상의 table

트리거

트리거(Trigger) 의 개요

- 트리거 실습
  - testDB에 테이블 생성

```
CREATE DATABASE IF NOT EXISTS testDB;
USE testDB;
CREATE TABLE IF NOT EXISTS testTbl (id INT, txt VARCHAR(10));
INSERT INTO testTbl VALUES(1, '레드벨벳');
INSERT INTO testTbl VALUES(2, '잇지');
INSERT INTO testTbl VALUES(3, '블랙핑크');
```

testDB라는 데이터베이스 만듦.
└여기에 test Tbl 만듦.

임시로 값을 집어넣음.

트리거

트리거(Trigger) 의 개요

- 트리거 실습
  - testTbl에 트리거 부착

```
DROP TRIGGER IF EXISTS testTrg;
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 실행 시 작동되는 코드들
END //
DELIMITER ;
```

트리거

트리거(Trigger) 의 개요

- 트리거 실습
  - 데이터 삽입, 수정, 삭제

```
SET @msg = '';
INSERT INTO testTbl VALUES(4, '마마무');
SELECT @msg;
UPDATE testTbl SET txt = '블핑' WHERE id = 3;
SELECT @msg;
DELETE FROM testTbl WHERE id = 4;
SELECT @msg;
```

→ global 변수인 msg라는 변수를 할당함.

└ 처음에는 빈 문자열

→ 삽입 후 msg 출력시 아무 내용 X

→ 변경 후 msg 출력시 아무 내용 X

→ 삭제 후 트리거 자동으로 동작해 msg 변수값의 내용이 달라짐

	@msg
▶	

	@msg
▶	

	@msg
▶	가수 그룹이 삭제됨

트리거

트리거의 종류

- AFTER 트리거
  - 테이블에 INSERT, UPDATE, DELETE 등의 작업이 일어났을 때 작동
  - 이름이 뜻하는 것처럼 해당 작업 후에(After) 작동

- BEFORE 트리거
  - BEFORE 트리거는 이벤트가 발생하기 전에 작동
  - INSERT, UPDATE, DELETE 세 가지 이벤트로 작동

트리거

트리거의 사용

- 트리거 문법

```
CREATE
    [DEFINER = user]
    TRIGGER trigger_name
    trigger_time trigger_event
    ON tbl_name FOR EACH ROW
    [trigger_order]
    trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

트리거

트리거의 사용

- AFTER 트리거의 사용
  - 예제 요구 사항
    - 회원 테이블에 update나 delete를 시도하면 수정 또는 삭제된 데이터를 별도의 테이블에 보관하고 변경된 일자와 변경한 사람을 기록

바로 지우거나 업데이트하는 게 아니라

백업

보관?...

구매등급 결정하는 방식. (gradeProc)
→ buyTbl에 구매내역 insert할 때 트리거 발동시켜 gradeProc 실행 → 고객이라 grade 자동으로 업데이트
user Tbl의
추가적으로 gradeProc 호출할 필요없이 자동으로 buyTbl 업데이트될 때마다 구매등급 update

트리거

트리거의 사용

```
DROP TRIGGER IF EXISTS gradeTrg;
DELIMITER //
CREATE TRIGGER gradeTrg  -- 트리거 이름
    AFTER  INSERT
    ON buytbl -- 트리거를 부착할 테이블
    FOR EACH ROW
BEGIN
    CALL gradeProc();
END //
DELIMITER ;
```

- AFTER 트리거의 사용
  - insert나 update 작업이 일어나는 경우, 변경되기 전의 데이터를 저장할 테이블 생성

```
USE sqlDB;
DROP TABLE buyTbl; -- 구매 테이블은 실습에 필요 없으므로 삭제
CREATE TABLE backup_userTbl
( userID  CHAR(8) NOT NULL PRIMARY KEY,
  name    VARCHAR(10) NOT NULL,
  birthYear   INT NOT NULL,
  addr        CHAR(2) NOT NULL,
  mobile1   CHAR(3),
  mobile2   CHAR(8),
  height    SMALLINT,
  mDate    DATE,
  modType  CHAR(2), -- 변경된 타입. '수정' 또는 '삭제'
  modDate  DATE, -- 변경된 날짜
  modUser  VARCHAR(256) -- 변경한 사용자
);
```

트리거

트리거의 사용

- AFTER 트리거의 사용
  - 변경(Update) 발생시 작동하는 backUserTbl_UpdateTrg 트리거 생성

```
DROP TRIGGER IF EXISTS backUserTbl_UpdateTrg;
DELIMITER //
CREATE TRIGGER backUserTbl_UpdateTrg  -- 트리거 이름
    AFTER UPDATE -- 변경 후에 작동하도록 지정
    ON userTBL -- 트리거를 부착할 테이블
    FOR EACH ROW
BEGIN
    INSERT INTO backup_userTbl VALUES( OLD.userID, OLD.name, OLD.birthYear,
        OLD.addr, OLD.mobile1, OLD.mobile2, OLD.height, OLD.mDate,
          '수정', CURDATE(), CURRENT_USER() );
END //
DELIMITER ;
```

트리거

트리거의 사용

- AFTER 트리거의 사용
  - 삭제(Delete) 발생시 작동하는 backUserTbl_DeleteTrg 트리거 생성

```
DROP TRIGGER IF EXISTS backUserTbl_DeleteTrg;
DELIMITER //
CREATE TRIGGER backUserTbl_DeleteTrg  -- 트리거 이름
    AFTER DELETE -- 삭제 후에 작동하도록 지정
    ON userTBL -- 트리거를 부착할 테이블
    FOR EACH ROW
BEGIN
    INSERT INTO backup_userTbl VALUES( OLD.userID, OLD.name, OLD.birthYear,
        OLD.addr, OLD.mobile1, OLD.mobile2, OLD.height, OLD.mDate,
        '삭제', CURDATE(), CURRENT_USER() );
END //
DELIMITER ;
```

트리거

트리거의 사용

- AFTER 트리거의 사용
  - 데이터 업데이트 및 삭제

```
UPDATE userTbl SET addr = '몽고' WHERE userID = 'JKW';
DELETE FROM userTbl WHERE height >= 177;
```

  - 수정 또는 삭제된 내용이 잘 보관되어 있는지 결과 확인

```
SELECT * FROM backup_userTbl;
```

userID	name	birthYear	addr	mobile1	mobile2	height	mDate	modType	modDate	modUser
JKW	조관우	1965	경기	018	9999999	172	2010-10-10	수정	2020-03-17	root@localhost
KKH	김경호	1971	전남	019	3333333	177	2007-07-07	삭제	2020-03-17	root@localhost
LJB	임재범	1963	서울	016	6666666	182	2009-09-09	삭제	2020-03-17	root@localhost
LSG	이승기	1987	서울	011	1111111	182	2008-08-08	삭제	2020-03-17	root@localhost
SSK	성시경	1979	서울	NULL	NULL	186	2013-12-12	삭제	2020-03-17	root@localhost

트리거

트리거의 사용

<AFTER DELETE로 트리거 작동>

delete

한꺼번에 지울 때는 트리거가 발동됨.

- AFTER 트리거의 사용
  - 테이블의 모든 행 데이터 삭제
  - 혹은 DROP TABLE 시 트리거 동작 안 함. (트리거 DML에 대해서만 동작함)
  - DELETE 대신 TRUNCATE TABLE문 사용

```
TRUNCATE TABLE userTbl;
```

  - 백업 테이블 확인

```
SELECT * FROM backup_userTbl;
```

userID	name	birthYear	addr	mobile1	mobile2	height	mDate	modType	modDate	modUser
JKW	조관우	1965	경기	018	9999999	172	2010-10-10	수정	2020-03-17	root@localhost
KKH	김경호	1971	전남	019	3333333	177	2007-07-07	삭제	2020-03-17	root@localhost
LJB	임재범	1963	서울	016	6666666	182	2009-09-09	삭제	2020-03-17	root@localhost
LSG	이승기	1987	서울	011	1111111	182	2008-08-08	삭제	2020-03-17	root@localhost
SSK	성시경	1979	서울	NULL	NULL	186	2013-12-12	삭제	2020-03-17	root@localhost

  - **TRUNCATE TABLE로 삭제 시에는 트리거가 작동하지 않음, DELETE 트리거는 DELETE문에만 작동**

트리거

트리거의 사용

- AFTER 트리거의 사용
  - INSERT 트리거를 생성

```
DROP TRIGGER IF EXISTS userTbl_InsertTrg;
DELIMITER //
CREATE TRIGGER userTbl_InsertTrg  -- 트리거 이름
    AFTER INSERT -- 입력 후에 작동하도록 지정
    ON userTBL -- 트리거를 부착할 테이블
    FOR EACH ROW
BEGIN
    SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = '데이터의 입력을 시도했습니다. 귀하의 정보가 서버에 기록되었습니다.';
END //
DELIMITER ;
```

성공을 한게 아님.
성공을 하지 못하고 에러가 난 것.

시그널의 강제로 발생. (강제로 mysql에게 에러 발생)

5위 SQLSTATE라는

잡히기

트리거

트리거의 사용

- AFTER 트리거의 사용
  - 데이터 입력
    INSERT INTO userTbl VALUES('ABC', '에비씨', 1977, '서울', '011', '1111111', 181, '2019-12-25'); → 트리거가 동작함.

Output

Action Output

	Time	Action	Message	Duration / Fetch
1	20:11:44	INSERT INTO userT...	Error Code: 1644. 데이터의 입력을 시도했습니다 .귀하의 정보가 서버에 기록되었습니다.	0.000 sec

우리가 트리거 동작에서 강제로 에러코드 발생.

결국 우리가 트리거 안에 있는 내용을 성공을 한 게 아님. 개방 트리거 발생시킨 insert 작업까지 취소.

  - 경고 메시지가 출력된 후에, INSERT 작업은 롤백이 되고 userTbl에는 데이터가 삽입되지 않음

일종의 Transaction 형태로 동작
(모두 실행이 완료하게 되거나 하나도 실행하지 않는 상태가 되거나)

트리거

트리거의 사용

- AFTER 트리거의 사용
  - TRUNCATE TABLE 테이블이름

    비슷한 용도로 사용. but 트리거 한정하면 truncate는 트리거 작동시키지 않는다는 차이점.
    - DELETE FROM 테이블이름문과 동일한 효과
    - DML문이 아니라 트리거를 작동시키지 않음

  - SIGNAL SQLSTATE '45000' 문
    - 사용자가 오류를 시그널 강제로 발생시키는 함수
    - 사용자가 정의한 오류 메시지 출력
    - 사용자가 시도한 INSERT는 롤백 하나의 트랜잭션 형태로 동작.

트리거

트리거의 사용

- 트리거가 생성하는 임시 테이블
  - INSERT, UPDATE, DELETE 작업이 수행되면 임시 사용하는 시스템 테이블
  - 이름은 'NEW'와 'OLD'

[그림 10-15] 트리거의 NEW와 OLD 테이블의 작동 개념

트리거

트리거의 사용

- BEFORE 트리거의 사용
  - 테이블에 변경이 가해지기 전 작동
  - BEFORE 트리거 활용 예
    - BEFORE INSERT 트리거를 부착해 놓으면 입력될 데이터 값을 미리 확인해서 문제가 있을 경우에 다른 값으로 변경 입력되면 안 되는 값 미리 걸러낼 수 있음.
  - BEFORE 트리거 실습
    - 값이 입력될 때, 출생년도의 데이터를 검사해서 데이터에 문제가 있으면 값을 변경시켜서 입력시키는 BEFORE INSERT 트리거 작성

트리거

트리거의 사용

- BEFORE 트리거의 사용
  - 트리거 생성

```
USE sqlDB;
DROP TRIGGER IF EXISTS userTbl_BeforeInsertTrg;
DELIMITER //
CREATE TRIGGER userTbl_BeforeInsertTrg  -- 트리거 이름
    BEFORE INSERT -- 입력 전에 작동하도록 지정
    ON userTBL -- 트리거를 부착할 테이블
    FOR EACH ROW
BEGIN
    IF NEW.birthYear < 1900 THEN
        SET NEW.birthYear = 0;
    ELSEIF NEW.birthYear > YEAR(CURDATE()) THEN
        SET NEW.birthYear = YEAR(CURDATE());
    END IF;
END //
DELIMITER ;
```

insert시 new라는 임시table에서 생년월일을 체크.

현재 년도

반대로 update는 어떻게?

set new.birthyear = old.birthyear.

after delete는 old가능한데 before도 가능한가?

after, before랑 old/new 사용여부 관계없는 듯

트리거

트리거의 사용

- BEFORE 트리거의 사용
  - 값 입력 (두 값 모두 출생년도에 문제 있음)

```
INSERT INTO userTbl VALUES
  ('AAA', '에이', 1877, '서울', '011', '1112222', 181, '2022-12-25');
INSERT INTO userTbl VALUES
  ('BBB', '비이', 2977, '경기', '011', '1113333', 171, '2019-3-25');
```

  - SELECT * FROM userTbl문으로 확인

userID	name	birthYear	addr	mobile1	mobile2	height	mDate
AAA	에이	0	서울	011	1112222	181	2022-12-25
BBB	비이	2020	경기	011	1113333	171	2019-03-25
BBK	바비킴	1973	서울	010	0000000	176	2013-05-05
EJW	은지원	1972	경북	011	8888888	174	2014-03-03
JKW	조관우	1965	경기	018	9999999	172	2010-10-10
	조용필		경기		44444		-04-04

트리거

트리거의 사용

- BEFORE 트리거의 사용
  - SHOW TRIGGERS문으로 데이터베이스에 생성된 트리거 확인

```
SHOW TRIGGERS FROM sqlDB;
```

어떤 이벤트에 발생하는지

Trigger	Event	Table	Statement	Timing	Created	sql_mode
userTbl_BeforeInsertTrg	INSERT	usertbl	BEGIN IF NEW.birthYear < 1900 THEN SET NEW.birthY...	BEFORE	2020-03-17 21:48:45.30	STRICT_TRANS_TABLES,NO_AUTO_(

어떤 table에 달려있는거.

  - 트리거 삭제

```
DROP TRIGGER userTbl_BeforeInsertTrg;
```

트리거 수정 (ALTER) 불가 - 한 번 만든 거 수정 불가. drop한다음 새로 만들어야함

트리거

기타 트리거에 관한 내용

- 다중 트리거 (Multiple Triggers)
  - 하나의 테이블에 동일한 트리거가 여러 개 부착되어 있는 것
    - ex) AFTER INSERT 트리거 한 개 테이블에 2개 이상 부착

      각각의 다른 일을 하는 트리거 붙일 수 있음.

- 중첩 트리거 (Nested Triggers) 하나의 트리거가 또 다른 테이블을 건드려 거기의 트리거를 동작시키고 거기가 또 다른 table의 트리거를 동작시킴.
  - 트리거가 또 다른 트리거를 작동시키는 것 → trigger를 기본적으로 씀으로써 트랜잭션화가 되기 때문에 유용

[그림 10-18] 중첩 트리거의 예

트리거

기타 트리거에 관한 내용

- 트리거의 작동 순서
  - 하나의 테이블에 여러 개의 트리거가 부착된 경우 트리거의 작동 순서 지정 가능

```
{ FOLLOWS | PRECEDES } other_trigger_name
```

뒤에 옴 이전에 옴.

나 이전에 무가 실행되어야 함 } 이전에 설정 가능
다음에 〃

트리거

기타 트리거에 관한 내용

- 중첩 트리거 작동 실습
  - 연습용 DB 생성

```
DROP DATABASE IF EXISTS triggerDB;
CREATE DATABASE IF NOT EXISTS triggerDB;
```

트리거

기타 트리거에 관한 내용

- 중첩 트리거 작동 실습
  - 테이블 생성

```
USE triggerDB;
CREATE TABLE orderTbl -- 구매 테이블
   (orderNo INT AUTO_INCREMENT PRIMARY KEY, -- 구매 일련번호
    userID VARCHAR(5), -- 구매한 회원 아이디
    prodName VARCHAR(5), -- 구매한 물건
    orderamount INT );  -- 구매한 개수
CREATE TABLE prodTbl -- 물품 테이블
   (prodName VARCHAR(5), -- 물건 이름
    account INT ); -- 남은 물건수량
CREATE TABLE deliverTbl -- 배송 테이블
   (deliverNo  INT AUTO_INCREMENT PRIMARY KEY, -- 배송 일련번호
    prodName VARCHAR(5), -- 배송할 물건
    account INT UNIQUE); -- 배송할 물건개수
```

트리거

기타 트리거에 관한 내용

- 중첩 트리거 작동 실습
  - 데이터 입력 재고를 최초에 이렇게 세팅

재고

```
INSERT INTO prodTbl VALUES('사과', 100);
INSERT INTO prodTbl VALUES('배', 100);
INSERT INTO prodTbl VALUES('귤', 100);
```

트리거

기타 트리거에 관한 내용

- 중첩 트리거 작동 실습
  - 구매 테이블(orderTbl)과 물품 테이블(prodTbl)에 트리거 부착

```
-- 물품 테이블에서 개수를 감소시키는 트리거
DROP TRIGGER IF EXISTS orderTrg;
DELIMITER //
CREATE TRIGGER orderTrg  -- 트리거 이름
    AFTER  INSERT
    ON orderTBL -- 트리거를 부착할 테이블
    FOR EACH ROW
BEGIN
    UPDATE prodTbl SET account = account - NEW.orderamount
        WHERE prodName = NEW.prodName ;
END //
DELIMITER ;
```

물건의 갯수

insert구나 new의 새롭게 입력된 거 갖고 가옴.

from table의 도리가 물품 table에 update를 함.

트리거

기타 트리거에 관한 내용

- 중첩 트리거 작동 실습
  - 구매 테이블(orderTbl)과 물품 테이블(prodTbl)에 트리거 부착

```
-- 배송 테이블에 새 배송 건을 입력하는 트리거
DROP TRIGGER IF EXISTS prodTrg;
DELIMITER //
CREATE TRIGGER prodTrg  -- 트리거 이름
    AFTER  UPDATE
    ON prodTBL -- 트리거를 부착할 테이블
    FOR EACH ROW
BEGIN
    DECLARE orderAmount INT; 변수 선언
    -- 주문 개수 = (변경 전의 개수 - 변경 후의 개수)
    SET orderAmount = OLD.account - NEW.account;
    INSERT INTO deliverTbl(prodName, account)
        VALUES(NEW.prodName, orderAmount);
END //
DELIMITER ;
```

update되기 전

물품 table 트리거는 배달 table에 insert.

트리거 한번하면 자동한다. 트랜잭션 형태 보장 → 아니면

트리거

기타 트리거에 관한 내용

결론
목적은 트랜잭션 형태로 동작.(한번에 실행됨)
(만약 5개 주문했는데 자고 update안하면 배송시작됨 이상해지잖)
한번에 이거 다 실행되거나 아예 주문 자체가 성공하지 못하거나.

- 중첩 트리거 작동 실습
  - 고객이 구매한 데이터 입력

```
INSERT INTO orderTbl VALUES (NULL,'JOHN', '배', 5);
```

└인서트만 했는데 과정이 자동되네

  - 중첩 트리거가 잘 작동했는지 세 테이블 확인

한번에 업데이트됨.

```
SELECT * FROM orderTbl;
SELECT * FROM prodTbl;
SELECT * FROM deliverTbl;
```

orderTbl의 트리거 작동 → prodTbl update
prodTbl의 트리거 작동 → deliverTbl insert

insert 됨

orderNo	userID	prodName	orderamount
1	JOHN	배	5

update

prodName	account
사과	100
배	95
귤	100

insert

deliverNo	prodName	account
1	배	5

100-95

트리거

기타 트리거에 관한 내용

- 중첩 트리거 작동 실습
  - 배송 테이블(deliverTbl)의 열 이름을 변경해서 (3)번의 INSERT가 실패하도록 실습

    중첩되서 동작하다 마지막에 insert가 결국 실패함

    ```
    ALTER TABLE deliverTBL CHANGE prodName productName VARCHAR(5);
    ```

    prodTbl의 트리거 prodName 호출.

  - 데이터 입력

    insert

    ```
    INSERT INTO orderTbl VALUES (NULL, 'DANG', '사과', 9);
    ```

    중첩 트리거 모두 실행되다 여기서 실패

    오류 메시지:

    Error Code: 1054. Unknown column 'prodName' in 'field list'

트리거

기타 트리거에 관한 내용

- 중첩 트리거 작동 실습
  - 테이블 확인

```
SELECT * FROM orderTbl;
SELECT * FROM prodTbl;
SELECT * FROM deliverTbl;
```

orderNo	userID	prodName	orderamount
1	JOHN	배	5

3에 내역 x

prodName	account
사과	100
배	95
귤	100

3에도 update x

deliverNo	prodName	account
1	배	5

배달 내역에 포함도 안됨.

여기서 실패하였음
(3에서, 제품 table에는 이미 반영이 됐지만도 불구하고
다시 롤백이 여기 이뤄지도
신용정리 안돼 것처럼 동작 해줌.

- **데이터가 변경되지 않았음.**
- **(3)번의 INSERT가 실패하면 (1)번 INSERT, (2)번 UPDATE 모두 롤백됨**

THANK YOU

무단 복제 및 배포를 금지합니다.