**1. 저장 프로시저의 의미와 작성 방법**

**1. 저장 프로시저란?**

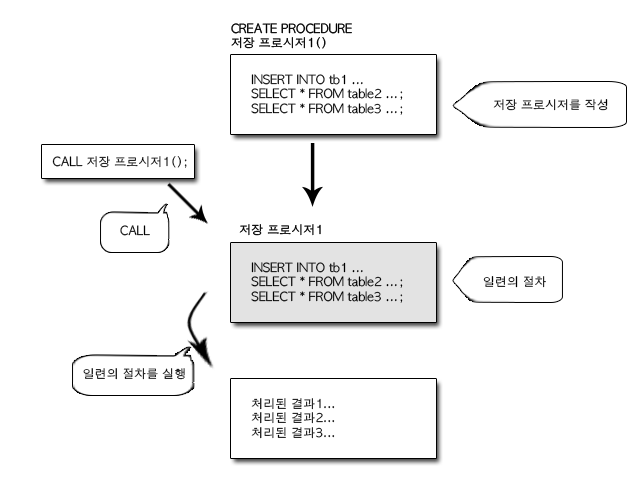
1.1 이용할 수 있는 버전

저장 프로시저를 학습하기 전에 MySQL의 버전을 확인하도록 합니다. 저장 프로시저는 MySQL 버전 5.0 이상에서 사용할 수 있습니다. 이전 버전에서는 사용할 수 없으니 사전에 확인하기 바랍니다.

1.2 저장 프로시저란 무엇인가?

여러 SQL 문을 하나의 SQL 문처럼 정리하여 'CALL ✕ ✕'라는 명령으로 실행할 수 있게 만든 것을 **저장 프로시저**(Stored Procedure)라고 합니다. Stored는 '저장하다', Procedur는 '절차'라는 의미입니다. 즉, 저장 프로시저는 일련의 절차를 정리해서 저장한 것입니다.

사전에 준비 둔 많은 명령을 자동으로 실행할 수 있기 때문에, 작업의 효율성도 높일 수 있습니다. 단, 중요한 데이터가 축적된 데이터베이스에서 제대로 검증되지 않은 저장 프로시저를 실행하는 것은 매우 위험합니다.



얼핏 보면 어려워 보이는 저장 프로시저이지만, 일단 사용 방법을 익혀두면 매우 편리합니다. 예를 들어, SELECT 문을 3번이나 입력하는 것이 번거롭다고 느껴질 때, CALL 한번이면 가볍게 해결할 수 있습니다.

다음은 이번에 설명할 저장 프로시저의 본체입니다.

SELECT \* FROM tb;

SELECT \* FROM tb1;

평범한 SQL 문을 2번 입력한 것뿐입니다. 이 예에서는 2개의 SQL 문으로 구성되어 있지만, 몇 개의 명령문을 입력해도 상관없습니다. 또한, 변수를 사용하거나 IF나 CASE로 분기 조건을 설정하거나, WHILE이나 REPEAT로 반복 처리를 할 수도 있습니다.

**2. 저장 프로시저 활용하기**

2.1 저장 프로시저 만들기

저장 프로시저를 작성할 때에는 다음과 같이 CREATE PROCEDURE라는 명령을 사용합니다.

**저장 프로시저 작성하기**

|  |
| --- |
| CREATE PROCEDURE 저장\_프로시저\_이름()  BEGIN  SQL 문1  SQL 문2  END |

**BEGIN**에서 **END**까지가 저장 프로시저의 본체입니다.

시작 부분에 BEGIN을, 끝 부분에 END를 붙여 저장 프로시저의 명령 범위를 명확히 하고 있습니다.

저장 프로시저의 본체는 '평범한 SQL 문'입니다. 그렇기 때문에, 당연히 명령문의 마지막에는 쌍반점(;)을 입력해야 합니다. 즉, 앞에서 소개한 저장 프로시저에서 본체 부분은 다음과 같이 기술해야 합니다.

BGIN

SELECT \* FROM tb;

SELECT \* FROM tb1;

END

하지만 이렇게 되면 저장 프로시저를 작성하는 중간에 쌍반점(;)을 입력하게 되는데, 그렇게 되면 저장 프로시저가 완성되지 않은 상태에서 CREATE PROCEDURE 명령이 실행되고 맙니다. MySQL 콘솔창은 쌍반점(;)이 입력되면 어떤 경우에서건 일단 쌍반점(;) 이전 단계까지 명령문을 실행하게 됩니다.

**구분 문자(;) 변경하기**

명령분이 완성되지 않은 상태에서 실행되면 곤란합니다. 저장 프로시저에서 END를 입력하고 나서 CREATE PROCEDURE 명령이 실행되도록 환경을 변경해야 합니다.

그러려면 저장 프로시저를 작성하기 전에 구분 문자를 쌍반점(;)이 아닌 다른 문자로 변경해 둡니다. 일반적으로는 //을 사용합니다.

구분 문자를 //으로 변경할 때에는 **DELIMITER** 명령을 사용합니다.

**구분 문자를 //으로 변경하기**

DELIMITER //

구분 문자를 //으로 변경해 두면 저장 프로시저를 작성하는 도중에 쌍반점(;)을 입력해도 문제없습니다. 그리고 END 뒤에 //을 입력하면 CREATE PROCEDURE 명령이 실행됩니다. 물론, 저장 프로시저를 모두 작성했으면 DELIMITER ;로 구분 문자를 원래대로 되돌려 놓습니다.

그럼, 예를 들어 설명하겠습니다. SELECT \* FROM tb;와 SELECT \* FROM tb1;을 실행하는 저장 프로시저 pr1을 생성하겠습니다.

다음 명령을 실행합니다.

DELIMITER //

CREATE PROCEDURE pr1()

BEGIN

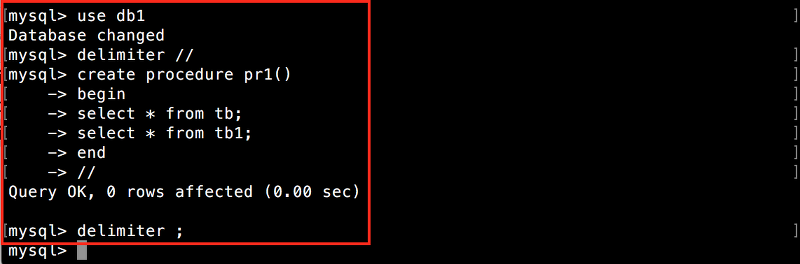
SELECT \* FROM tb;

SELECT \* FROM tb1;

END

//

DELIMITER ;



맨 마지막의 DELIMITER ;는 구분 문자를 원래대로 되돌려 놓는 명령입니다. 구분 문자를 쌍반점(;)으로 되돌려 놓는 것을 잊지 않도록 합니다.

또한, 저장 프로시저 이름 뒤에는 반드시 괄호()를 붙입니다. 저장 프로시저에 인수를 대입하는 방법에 대해서는 다음 절에서 소개하겠지만, 인수가 없더라도 괄호는 생략하면 안됩니다.

**2.2 저장 프로시저 실행하기**

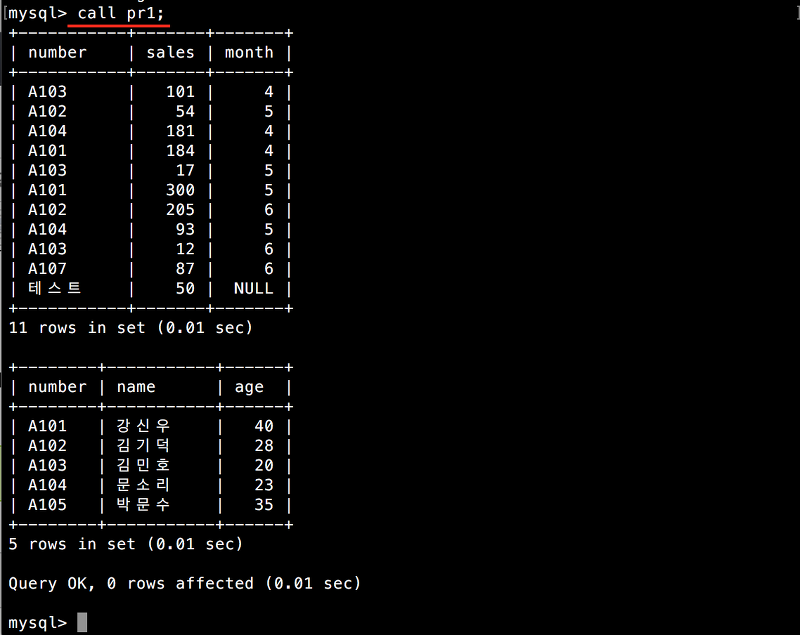
이제 저장 프로시저 pr1이 생성되었습니다. 그럼, CALL이라는 명령을 사용해서 저장 프로시저를 실행해 보겠습니다.

**저장 프로시저 실행하기**

CALL 저장\_프로시저\_이름;

앞에서 작성한 pr1을 호출해 보겠습니다.

CALL pr1;



SELECT \* FROM tb;와 SELECT \* FROM tb1;이 연속해서 실행되었습니다.

**2.3 설정한 값 이상인 레코드만 표시하는 저장 프로시저 만들기**

계속해서 인수를 대입해서 실행하는 저장 프로시저를 만들어 보겠습니다. 처리 하고자 하는 데이터를 괄호() 안에 대입해서 저장 프로시저를 실행합니다.

다음은 저장 프로시저에 인수를 대입하는 방법입니다.

**저장 프로시저에 인수 대입하기**

PROCEDURE 저장\_프로시저\_이름(인수\_이름 자료형);

이번에는 간단하게 대입한 값 이상의 매출이 있는 레코드를 표시하는 기본적인 저장 프로시저를 만들어 보겠습니다. 예를 들어, 저장 프로시저 pr에 정수형의 인수 d를 대입한다면, PROCEDURE pr(d INT)라고 기술합니다. 인수로 설정한 d는 일반적인 숫자와 마찬가지로 SQL 문 안에 기술합니다.

예를 들어, 다음과 같은 처리를 한다고 사정해 봅시다.

◼︎ 테이블 tb에서 sales가 인수 d이상인 레코드 표시

그러면 다음과 같이 작성합니다.

SELECT \* FROM tb WHERE sales>=d;

이제 정수형의 인수 d에 값을 대입해서 실행하면, 테이블 tb에서 매출 sales가 d이상인 레코드만 표시하는 저장 프로시저 pr2를 만들어 보겠습니다.

다음 명령을 실행합니다.

DELIMITER //

CREATE PROCEDURE pr2(d INT)

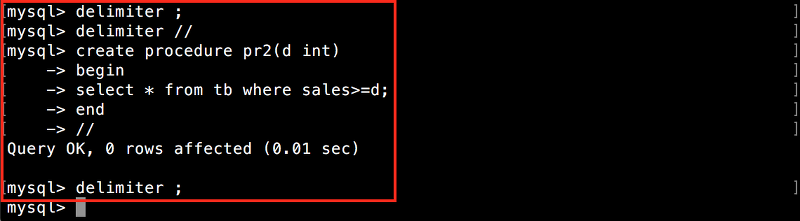
BEGIN

SELECT \* FROM tb WHERE sales>=d;

END

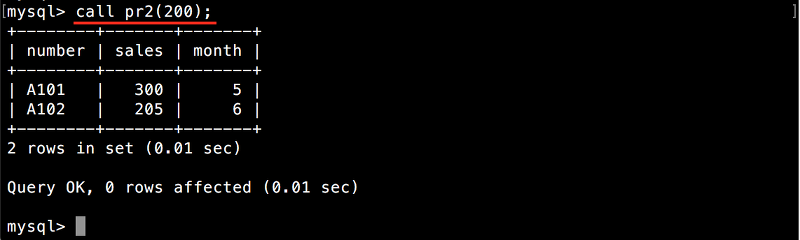
//

DELIMITER ;



이번에는 인수 200을 대입해서 프로시저 pr2를 실행해 보겠습니다. 괄호 () 안에 200을 입력해서 CALL을 실행합니다.

CALL pr2(200);



200 이상의 매출이 있는 레코드만 표시되었습니다.

**인수 IN을 대입하는 예**

위의

DELIMITER //

CREATE PROCEDURE pr2(d INT)

BEGIN

SELECT \* FROM tb WHERE sales>=d;

END

//

DELIMITER ;

을 입력할 때, 인수 앞에 IN 을 입력해도 결과는 같습니다(IN d INT). 반대로 처리한 결과를 인수에 대입하는 경우에는 OUT을 입력합니다.

DELIMITER //

CREATE PROCEDURE pr2(IN d INT)

BEGIN

SELECT \* FROM tb WHERE sales>=d;

END

//

DELIMITER ;

**3. 저장 프로시저의 내용 표시하고 삭제하기**

저장 프로시저의 내용을 표시하는 방법과 삭제하는 방법을 설명하겠습니다.

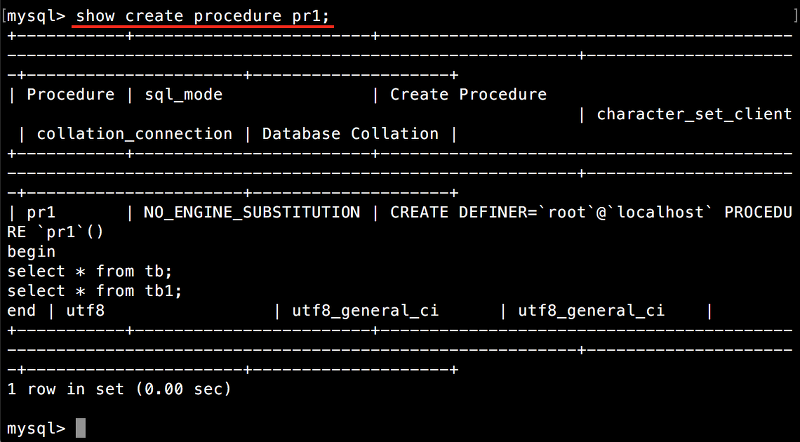
3.1 저장 프로시저의 내용 표시하기

작성한 저장 프로시저의 내용을 표시할 때에는 다음 명령을 실행합니다.

**저장 프로시저의 내용 표시하기**

SHOW CREATE PROCEDURE 저장\_프로시저\_이름;

다음의 저장 프로시저 pr1의 내용을 표시한 예입니다.



3.2 저장 프로시저 삭제하기

저장 프로시저를 삭제할 때는 데이터베이스나 테이블, 뷰를 삭제할 때와 마찬가지로 DROP 명령을 사용합니다.

**저장 프로시저 삭제하기**

DROP PROCEDURE 저장\_프로시저\_이름;

다음은 저장 프로시저 pr1을 삭제하는 예입니다.



**2. 저장 함수의 의미와 작성 방법**

4. 저장 함수란?

4.1 이용할 수 있는 버전

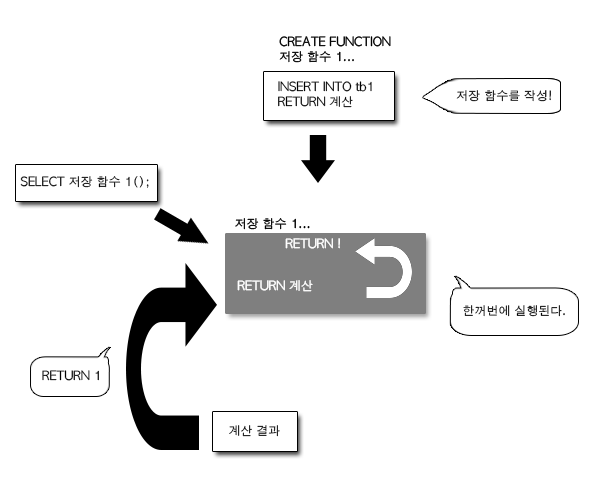
저장 함수도 MySQL 버전 5.0 이상에서 사용할 수 있습니다. 이전 버전에서는 사용할 수 없으니 사전에 확인하기 바랍니다.

4.2 저장 함수란 무엇인가?

**저장 함수**는 저장 프로시저와 거의 흡사합니다. 저장 프로시저와 유일하게 다른 점은 실행 했을 때 값을 반환한다는 점입니다.

저장 함수(Stored Function)는 이름 그대로 함수 역할을 합니다. 8장의 2.3 각종 정보를 표시하는 함수에는 여러 함수가 있다고 설명했는데, 저장 함수를 사용하면 자신만의 함수를 만들 수가 있습니다. 저장 함수는 **사용자 정의 함수**라고도 합니다.

저장 함수



저장 함수가 반환한 값은 SELECT나 UPDATE 등의 명령을 이용해서 일반 함수와 마찬가지로 이용할 수 있습니다.

다음은 저장 함수를 작성하는 구문입니다.

**저장 함수 작성하기**

CREATE FUNCTION 저장\_함수\_이름(인수 자료형) RETURNS 반환값의\_자료형

BEGIN

SQL 문...

RETURN 반환값 식

END

저장 프로시저와 마찬가지로 괄호() 안에 인수를 대입할 수 있습니다. 인수를 대입하지 않아도 괄호는 입력해야 합니다.

예를 들어, 'fu'라는 이름의 저장 함수를 만든다면, 저장 함수의 RETUREN ✕ ✕의 ✕ ✕는 fu() 함수가 반환하는 값입니다.

5. 저장 함수 활용하기

5.1 저장 함수로 표준 체중 계산하기

설명만으로는 저장 함수에 대해 제대로 이해하기 어려울 것입니다. 이해를 돕고자 '표준 체중'을 계산하는 연습문제를 풀어 보겠습니다.

'BMI(body Mass Index) = 22'가 표준 체중이라고 했을 때, 다음과 같은 식이 성립합니다.

■ 표준 체중 = 신장(cm 단위) Х 신장(cm 단위) Х 22 / 10000

이 식을 이용해서 저장 함수 fu1()을 만들어 보겠습니다. 인수에는 cm 단위의 신장(정수형)을 'height'라는 이름으로 설정하겠습니다. 다음은 저장 함수 fu1()에 INT형의 인수를 대입하는 예입니다.

CREATE FUNCTION fu1(height INT)

저장 함수는 그 자체가 값을 반환합니다. 그렇기 때문에, 이 저장 함수가 반환할 값의 자료형을 지정해야 합니다. 이번에 저장 함수 fu1()이 반환하는 값은 소수점 이하를 포함하는 표준 체중입니다. 따라서 저장 함수 fu1()이 반환할 값의 자료형은 DOUBLE형으로 정하겠습니다. 결과는 다음과 같습니다.

CREATE FUNCTION fu1(height INT) RETURNS DOUBLE

이 저장 함수는 신장 height를 2번 곱한 값에 22를 곱하고 10,000으로 나누고 나서, 그 결괏값을 RETURN으로 돌려줍니다. 다음은 이 내용을 작성한 예입니다.

RETURN height \* height \* 22 / 1000;

그럼, 이제 표준 체중을 계산하는 함수를 만들어 보겠습니다. cm 단위의 신장을 인수 height에 대입하면 표준 체중을 반환하는 저장 함수 fu1()을 만듭니다.

다음 명령을 실행한다.

DELIMITER //

CREATE FUNCTION fu1(height INT) RETURNS DOUBLE

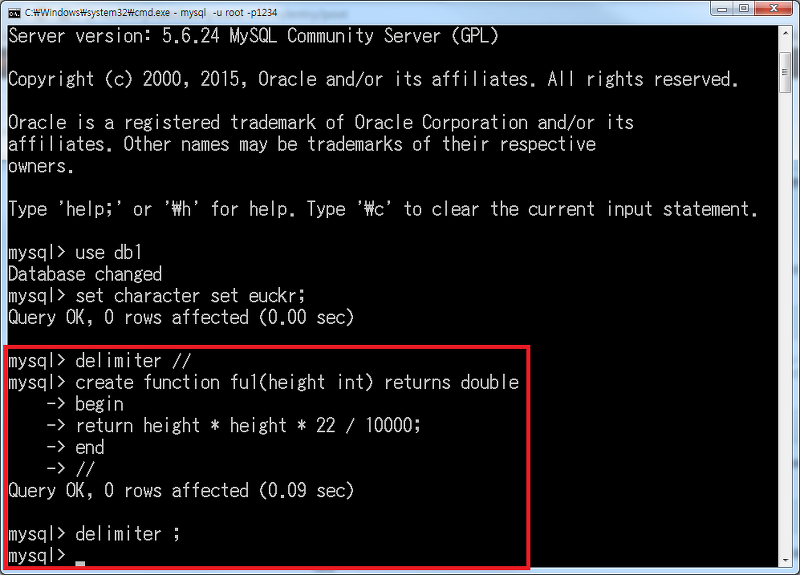
BEGIN

RETURN height \* height \* 22 / 10000;

END

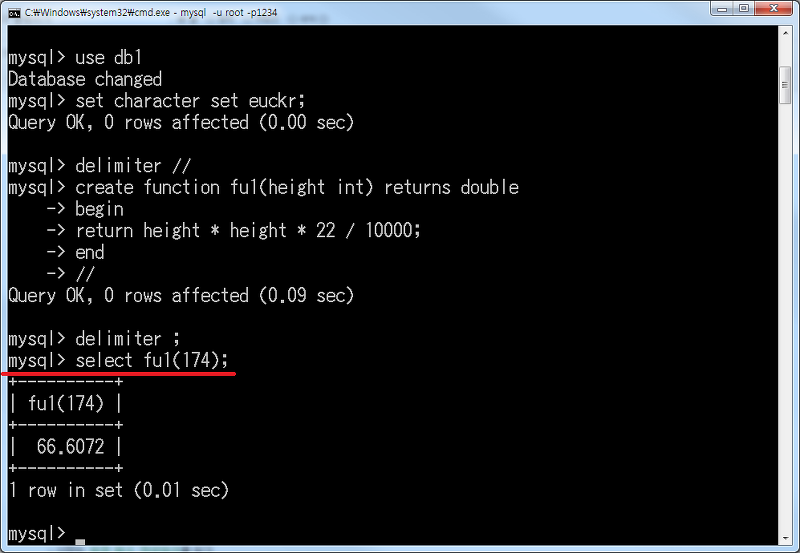
//

DELIMITER ;



그럼 신장 174cm로 계산을 해보겠습니다. 이번에는 fu1()이 함수로서 값을 반환합니다. 따라서 CALL을 사용하지 않고, SELECT fu1()의 값을 표시 합니다. 괄호 안에 대입할 인수의 값은 174입니다.

SELECT fu1(174);



5.2 레코드의 평균값을 반환하는 저장 함수

특정 테이블에서 특정 칼럼의 평균값을 반환하는 함수를 저장 함수로 만들어 보겠습니다. 이런 함수는 미리 만들어 두면 꽤 편리하게 사용할 수 있습니다.

다음은 완성된 저장 함수입니다. 테이블 tb에서 칼럼 sales의 평균을 반환하는 저장 함수 fu2()를 생성하는 예입니다.

CREATE FUNCTION fu2() RETURNS DOUBLE - ①

BEGIN

DECLARE r DOUBLE; - ②

SELECT AVG(sales) INTO r FROM tb; - ③

RETURN r; - ④

END

프로그램 언어에 대한 경험이 적은 분은 변수를 다루는 데 어려움이 있을지도 모르겠지만, 일단 이대로 입력합니다. 변수의 의미에 대해서는 이어서 설명하겠습니다.

① 부분은 앞에서 설명했듯이 저장 함수 fu2()가 반환하는 값의 자료형을 DOUBLE로 지정하고 있습니다.

CREATE FUNCTION fu2() RETURNS DOUBLE

평균값은 AVG() 함수를 사용해서 SELECT AVG()로 구합니다. 이 값은 일단 변수에 대입해야 합니다. 변수는 '값을 보관하는 상자'입니다. 변수를 사용하려면 미리 **DECLARE**로 정의해야 합니다.

**DECLARE로 변수 정의하기**

DECLARE 변수\_이름 자료형;

여기에서는 평균을 대입하는 변수 이름을 r이라고 하겠습니다. 변수 r에는 평균값이 저장 됩니다. 따라서 자료형은 소수점 이하의 값에 대응할 수 있는 DOUBLE형으로 설정합니다. r이라는 변수를 DOUBLE형으로 정의한 부분이 ②입니다.

DECLARE r DOUBLE;

테이블 tb에서 칼럼 sales의 평균값을 추출하는 SQL문은 다음과 같습니다.

SELECT AVG(sales) FROM tb;

③에서는 AVG(sales)를 r 에 대입할 때 INTO를 사용합니다.

SELECT AVG(sales) INTO r FROM tb;

이렇게 하면 변수 r에 평균값이 대입됩니다. 이 값을 ④의 RETURN에서 저장 함수의 결괏값으로 반환하고 있습니다.

RETURN r;

그럼, 실제로 테이블 tb에서 칼럼 sales의 평균을 DOUBLE형으로 반환하는 저장 함수 fu2()를 만들어 보겠습니다.

다음 명령을 실행합니다.

DELIMITER //

CREATE FUNCTION fu2() RETURNS DOUBLE

BEGIN

DECLARE r DOUBLE;

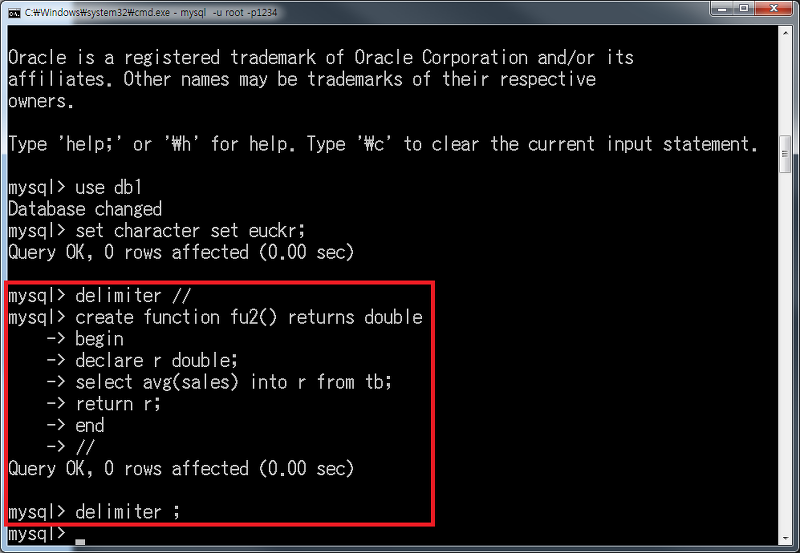
SELECT AVG(sales) INTO r FROM tb;

RETURN r;

END

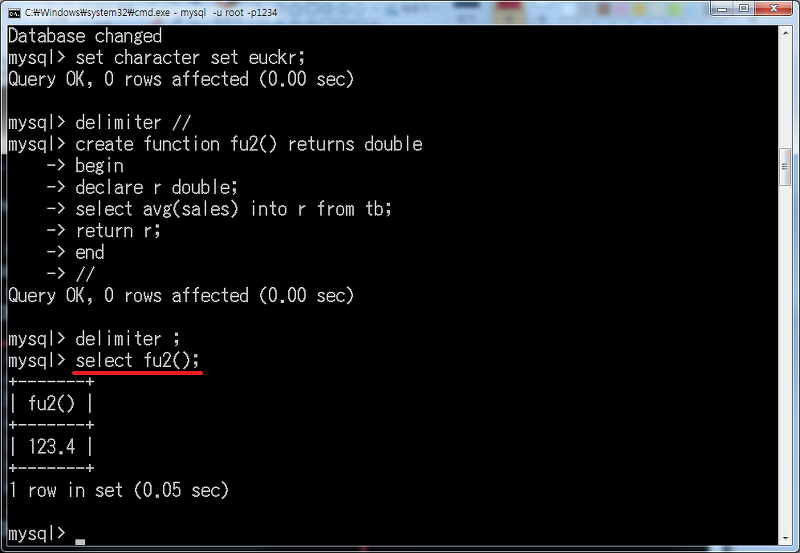
//

DELIMITER ;



작성했으면 SELECT 명령으로 평균값을 표시해 보겠습니다.

SELECT fu2();



평균값 123.4가 표시되었습니다.

5.3 저장 함수의 내용 표시하고 삭제하기

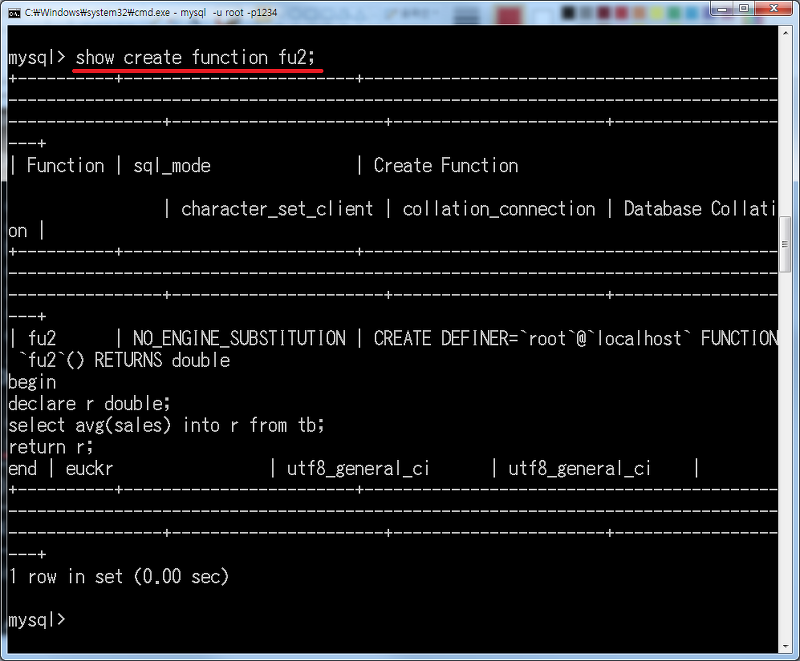
저장 함수의 내용을 표시하거나 삭제하는 방법은 저장 프로시저와 같습니다.

**저장 함수 삭제하기**

DROP FUNCTION 저장\_함수\_이름;

**저장 함수의 내용 표시하기**

SHOW CREATE FUNCTION 저장\_함수\_이름;



**4. 트리거의 의미**

6. 트리거란?

6.1 이용할 수 있는 버전

트리거도 MySQL 5.0 이상 버전에서 사용할 수 있습니다. 이전 버전에서는 이용할 수 없으니 사전에 확인하기 바랍니다.

6.2 트리거란 무엇인가?

트리거(trigger)는 테이블에 대해 어떠한 처리를 실행하면 이에 반응하여 설정해 둔 명령이 자동으로 실행되는 구조를 말합니다. 이때, 이러한 과정이 총의 방아쇠를 당기는 것과 같다하여 트리거(방아쇠)라고 합니다.

INSERT나 UPDATE,DELETE 등의 명령이 실행될 때, 사전에 트리거로 설정해 놓은 기능도 함께 실행할 수가 있습니다. 예를 들어, 테이블의 레코드를 변경하면 그것을 계기로 변경한 내용을 다른 테이블에 기록하도록 트리거를 작성할 수 있습니다.

트리거는 처리를 기록하거나, 만약 처리가 실패했을 때를 대비해서 만들어 놓으면 좋습니다.

트리거는 강력한 기능이지만, 설명을 듣는 것만으로는 어떤 장점이 있는지 잘 와 닿지 않을것입니다. 일단, 트리거를 직접 만들어서 체험해 보겠습니다.

여기에서는 테이블에 있는 레코드를 삭제하면, 삭제한 레코드가 다른 테이블에 복사되는 트리거를 만들어 보겠습니다. 테이블 tb1에 DELETE FROM tb1... 명령이 실행되면 삭제한 레코드는 모두 테이블 tb1M에 삽입됩니다. 이제 삭제한 레코드는 언제든지 복원할 수 있습니다.

테이블 tb1의 칼럼 구조만 복사해서 테이블 tb1M을 미리 만들어 둡니다.

CREATE TABLE tb1M LIKE tb1;

**5. 트리거로 추출할 수 있는 칼럼의 데이터 종류와 시점과 작성 방법**

7. 트리거 만들기

트리거는 INSERT나 UPDATE, DELETE 등의 명령이 실행되기 직전(BEFORE) 또는 직후(AFTER)에 호출되어 실행됩니다.

7.1 트리거가 호출되는 시점

트리거가 호출되는 시점은 다음 2가지입니다.

▶ 트리거가 호출되는 시점

|  |  |
| --- | --- |
| BEFORE | 테이블에 어떠한 처리를 하기 직전에 호출된다. |
| AFTER | 테이블에 어떠한 처리를 한 직후에 호출된다. |

또한, 이 테이블에서 어떠한 처리를 하기 직전과 직후의 값은 다음과 같이 'OLD.칼럼\_이름'과 'NEW.칼럼\_이름'으로 얻을 수 있습니다.

▶ 칼럼의 값

|  |  |
| --- | --- |
| OLD.칼럼\_이름 | 테이블에 어떠한 처리가 이루어 지기 직전의 칼럼값 |
| NEW.칼럼\_이름 | 테이블에 어떠한 처리가 이루어 진 직후의 칼럼 값 |

INSERT나 UPDATE, DELETE 명령이 실행되기 전의 칼럼 값은 'OLD.칼럼\_이름'으로, 실행한 후의 칼럼 값은 'NEW.칼럼\_이름'으로 추출할 수 있습니다.

단, 명령에 따라서는 칼럼 값을 추출할 수 있는 것과 없는 것이 있습니다. 다음 표에서 ◯표시가 추출할 수 있는 명령입니다.

트리거를 실행한 전후, 칼럼값의 추출 여부

|  |  |  |
| --- | --- | --- |
| **명령** | **실행 전(OLD.칼럼\_이름) ※ BEFORE 사용** | **실행 후(NEW.칼럼\_이름) ※ AFTER 사용** |
| INSERT | Х | ◯ |
| DELETE | ◯ | Х |
| UPDATE | ◯ | ◯ |

7.2 트리거 만들기

실제로 사용해 보는 것이 가장 이해가 빠를 것입니다. 다음은 트리거를 작성하는 방법입니다.

**트리거 작성하기**

CREATE TRIGGER 트리거\_이름 BEFORE(또는 AFTER) DELETE 등의 명령

ON 테이블\_이름 FOR EACH ROW

BEGIN

변경 전(OLD.칼럼\_이름) 또는 변경 후(NEW.칼럼\_이름)을 이용한 처리

END

트리거의 본체를 작성할 때 각 명령의 끝에느 쌍반점(;)을 입력합니다. 따라서, 저장 프로시저와 마찬가지로 미리 구분 문자를 //으로 변경해 둡니다.

그럼, 본격적으로 트리거를 만들어 보겠습니다. 테이블 tb1의 레코드를 삭제(DELETE)하고, 삭제한 레코드를 테이블 tb1M에 삽입하는 트리거 tr1을 작성합니다.

다음 명령을 실행합니다.

DELIMITER //

CREATE TRIGGER tr1 BEFORE DELETE ON tb1 FOR EACH ROW

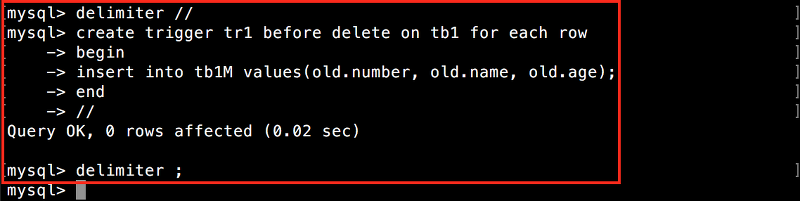
BEGIN

INSERT INTO tb1M VALUES(OLD.number, OLD.name, OLD.age);

END

//

DELIMITER ;

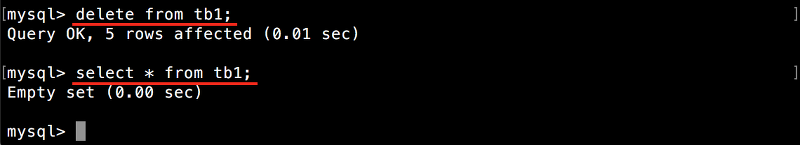


트리거가 완성되었습니다. 이 내용에 대해서는 이어서 설명하겠습니다. 일단, 트리거의 움직임을 확인해 보겠습니다.

이제 테이블 tb1에서 삭제한 레코드가 모두 테이블 tb1M에 추가되었을 것입니다. 몇 개의 레코드를 삭제해도 결과에는 변함이 없지만, 이번에는 레코드를 전부 삭제해 보겠습니다.

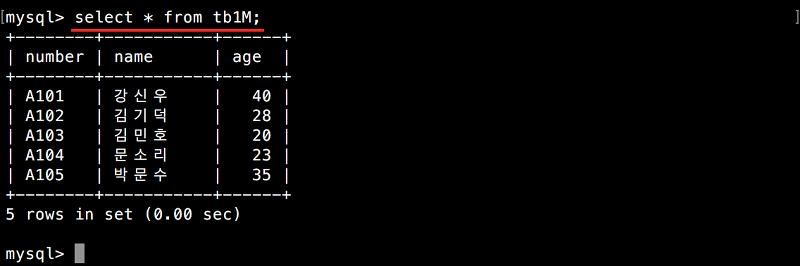
DELETE FROM tb1;

일단, SELECT \* FROM tb1;을 실행해서 레코드가 전부 삭제되었는지 확인해 보겠습니다.



'Empty set'이라고 표시되며, 테이블 tb1에는 레코드가 남아 있지 않습니다. 마지막으로, 설정한 트리거가 제대로 동작했는지 테이블 tb1M을 확인해 보겠습니다.

SELECT \* FROM tb1M;



예상했던 대로 삭제한 레코드가 모두 추가되었습니다(SELECT로 표시된 순서는 다를 수 있습니다). 트리거가 제대로 동작한 것을 확인했습니다. 만약 동작하지 않았다면 입력한 이력을 다시 한번 확인하기 바랍니다. 동작하지 않는 원인 중 가장 잦은 원인은 입력 오류입니다. 다음 절에서 처럼 SHOW 명령으로 내용을 확인할 수 있습니다.

만약, 트리거의 tr1의 내용에 틀린 부분이 있다면 DROP TRIGGER tr1;을 실행해서 트리거를 삭제하고, 다시 작성합니다.

그럼, 테이블 tb1M에 삽입한 레코드를 원래대로 tb1에 복원하도록 합니다. 레코드를 삽입하는 명령은 다음과 같습니다.

INSERT INTO tb1 SELECT \* FROM tb1M;

7.3 작성한 트리거의 내용

그럼 이제 트리거의 내용을 살펴보겠습니다.

CREATE TRIGGER tr1 BEFORE DELETE ON tb1 FOR EACH ROW

BEGIN

INSERT INTO tb1M VALUES(OLD.number, OLD.name, OLD.age);

END

//

먼저, 테이블 tb1에 대한 DELETE명령에 반응하는 트리거 tr1을 설정합니다. 삭제하기 직전(BEFORE)의 값을 INSERT하기 때문에 CREATE TRIGGER는 다음과 같이 작성합니다.

CREATE TRIGGER tr1 BEFORE DELETE ON tb1 FOR EACH ROW

레코드를 삭제하기 전(BEFORE)의 값(OLD.칼럼\_이름)을 추출해서 테이블 tb1M에 삽입 합니다. 테이블 tb1M의 칼럼은 number와 name, age이므로, OLD.number와 OLD.name, OLD.age가 삭제하기 직전의 칼럼 값이 됩니다.

이 칼럼 값은 INSERT 명령으로 테이블 tb1M에 삽입하므로 트리거의 본체는 다음과 같습니다.

INSERT INTO tb1M VALUES(OLD.number, OLD.name, OLD.age);

그리고 이 본체의 시작과 끝 부분에 BEGIN과 END를 입력합니다.

8. 트리거 확인하고 삭제하기

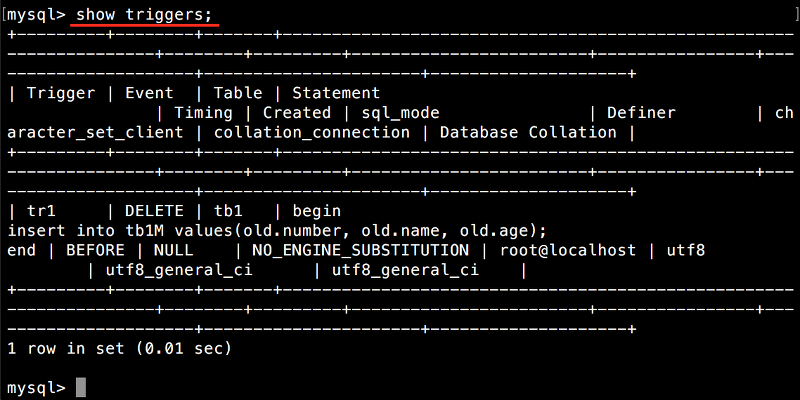
8.1 설정한 트리거 확인하기

트리거는 자동으로 실행됩니다. 따라서 의도하지 않은 동작이 발생하지 않도록 관리에 주의를 기울여야 합니다. 현재 설정된 트리거와 그 내용은 항상 파악해 두기 바랍니다.

현재 설정된 트리거를 확인할 때에는 **SHOW TRIGGERS** 명령을 사용합니다.

**트리거 확인하기**

SHOW TRIGGERS;



8.2 트리거 삭제하기

의도하지 않은 동작이 발생하지 않도록 필요 없는 트리거는 삭제해 둡니다.

**트리거 삭제하기**

DROP TRIGGER 트리거\_이름;

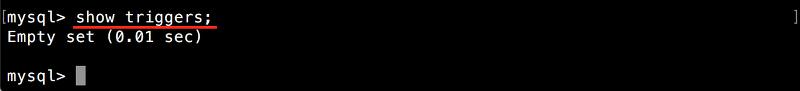
트리거 tr1을 삭제해 보겠습니다.

다음 명령을 실행합니다.

DROP TRIGGER tr1;



트리거 tr1이 삭제되었는지 SHOW TRIGGERS; 명령으로 확인해 보겠습니다.



저장 프로시저는 여러 처리를 한꺼번에 실행하는 것이라고 배웠습니다. 서버에서 이루어지는 처리를 저장 프로시저로 정리해 두면 클라이언트와 서버 간에 오가는 작업을 줄일 수 있습니다. 즉, 저장 프로시저를 이용하면 처리에 필요한 전체속도를 높일 수 있습니다. 또한, 일련의 처리를 정리해 둠으로써 작업에 실수를 줄일 수도 있습니다.

체크!

★ CREATE PROCEDURE로 저장 프로시저를 생성하고 CALL로 실행할 수 있다.

★ 구부 문자를 변경할 수 있다.

★ 인수를 입력받는 저장 프로시저를 작성할 수 있다.

★ 저장 함수를 작성할 수 있다.

★ SELECT로 얻은 값을 반환하는 저장 함수를 만들 수 있다.

★ CREATE TRIGGER로 트리거를 생성할 수 있다.

★ BEFORE나 AFTER로 실행 전후의 칼럼 값을 추출할 수 있다.

<https://recoveryman.tistory.com/190?category=595002>