MySQL

Процедуры и триггеры

Процедуры

Введение

В предыдущих разделе мы подробно поговорили о функциях, которые позволяют обрабатывать какие-либо аргументы и возвращают нам результат этой обработки в виде единственного значение (текста, числа, даты и т.д.). Однако нам может понадобиться возвращение не одного значения, а целого набора значений. Кроме того, иногда требуется передать аргумент, значение которого в функции должно поменяться, а после исполнения функции мы должны получить этот аргумент без возвращения. К сожалению, посредством функций этого добиться невозможно. Однако в MySQL присутствует инструмент, который позволяет решить обозначенные проблемы - это процедуры.

Остальные преимущества процедур заключаются прежде всего в компактности - одна команда позволяет вызвать сложный сценарий, который содержится в процедуре, что позволяет избежать пересылки данных через сеть сотен команд. Помимо этого, при первом использовании для процедуры составляется оптимизированный план доступа к данным, что ускоряет ее выполнение в последующие разы.

Подготовка (создание таблиц и добавление данных)

```
-- создание таблицы
CREATE TABLE animals (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    breed VARCHAR(255) NOT NULL,
    age INT UNSIGNED NOT NULL,
    food_kg_per_day DECIMAL(20,3)
);

-- заполнение таблицы данными
INSERT INTO animals VALUES
(1, 'Shere Khan', 'Tiger', 8, 12),
(2, 'Bagheera', 'Panther', 6, 7),
(3, 'Father', 'Wolf', 10, 5),
(4, 'Raksha', 'Wolf', 11, 6),
(5, 'Akela', 'Wolf', 15, 7),
(6, 'Baloo', 'Bear', 20, 10),
(7, 'Kaa', 'Python', 7, 2),
(8, 'Hathi', 'Elephant', 42, 15);
```

Подготовка (содержимое таблицы)

Содержимое таблицы **animals**

id	name	breed	age	food_kg_per_day
1	Shere Khan	Tiger	8	12
2	Bagheera	Panther	6	7
3	Father	Wolf	10	5
4	Raksha	Wolf	11	6
5	Akela	Wolf	15	7
6	Baloo	Bear	20	10
7	Kaa	Python	7	2
8	Hathi	Hathi	42	15

Основы процедур (различия с функциями)

Процесс создания процедур похож на процесс создания функций, но есть четыре основных отличия:

- создание непосредственно процедуры начинается с команды **CREATE PROCEDURE**
- не надо указывать, какой тип данных будет возвращать процедура
- перед каждым аргументом процедуры необходимо указывать его предназначение с помощью особых ключевых слов - **IN** (аргумент используется только внутри процедуры), **OUT** (значение аргумента которого может быть получено после того, как процедура была выполнена) и **INOUT** (значение аргумента может быть использовано внутри процедуры, его значение можно получить после исполнения функции). Но аргументы надо указывать только если они необходимы.
- не надо указывать характеристику **DETERMINISTIC** (т.е. не надо явно обозначать, что будет возвращены одни и те же данные при одних и тех же аргументах)

Основы процедур (визуальная схема)

```
CREATE
    [DEFINER = user]
    PROCEDURE [IF NOT EXISTS] имя_процедуры
([аргумент_процедуры [,...]])
    [характеристики ...] тело_процедуры

аргумент_процедуры:
    [ IN | OUT | INOUT ] название тип

тип:
    любой тип MySQL

характеристики: {
    COMMENT 'string'
    LANGUAGE SQL
    { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL
DATA }
    SQL SECURITY { DEFINER | INVOKER }
}

тело_процедуры:
    Инструкции SQL без ошибок
```

Базовое создание процедур

Попробуем создать нашу первую процедуру. Не будем использовать аргументы, просто получим имена всех зверей из таблицы **animals**. Обратим внимание, что получение данных (по сути - возвращение таблицы из процедуры) осуществляется с помощью конструкции **SELECT**.

```
DELIMITER |
CREATE PROCEDURE IF NOT EXISTS get_animal_names()
BEGIN
SELECT name FROM animals;
END |
DELIMITER ;
```

Вызов процедур

Вызовем нашу новую процедуру **get_animal_names** при помощи ключевого слова **CALL**. В ответ мы получим стандартную таблицу из имен зверей:

```
CALL get_animal_names();
```

name
Shere Khan
Bagheera
Father
Raksha
Akela
Baloo
Kaa
Hathi

Возвращение двух или более таблиц из процедуры (создание процедуры)

Интересный момент - из процедуры можно вернуть не одну, а две или более таблиц. Для этого всего навсего надо два (или более) раза применить **SELECT**. Напишем процедуру, которая, например, возвращает первую таблицу с именами, а вторую, отдельную таблицу - с видами зверей:

```
DELIMITER |

CREATE PROCEDURE IF NOT EXISTS get_animal_names_and_breeds()
BEGIN
SELECT name FROM animals;
SELECT DISTINCT breed FROM animals;
END |

DELIMITER ;
```

Возвращение двух или более таблиц из процедуры (вызов процедуры)

Вызовем процедуру **get_animal_names_and_breeds** и получим в ответ две таблицы:

```
CALL get_animal_names_and_breeds();
```

name
Shere Khan
Bagheera
Father
Raksha
Akela
Baloo
Kaa
Hathi

breed
Tiger
Panther
Wolf
Bear
Python
Elephant

Удаление процедур

Удаляется процедура стандартно - при помощи команды **DROP PROCEDURE** после чего идет название удаляемой процедуры. Если мы не уверены, что процедура существует, и мы хотим избежать потенциальной ошибки, можем применить конструкцию **IF EXISTS**:

```
-- удаление процедуры без проверки
DROP PROCEDURE get_animal_names;

-- удаление существующей процедуры с проверкой
DROP PROCEDURE IF EXISTS get_animal_names_and_breeds;

-- удаление несуществующей процедуры с проверкой
DROP PROCEDURE IF EXISTS unexisting_procedure;
```

Использование аргументов в процедурах (IN) (создание процедуры)

Теперь мы попробуем применить наши знания, чтобы создать процедуру, которая принимает аргумент с назначением **IN** - только для использования его внутри процедуры. Будем получать все зверей определенного вида - строку с видом будем передавать в процедуру в качестве аргумента.

```
DELIMITER |

CREATE PROCEDURE IF NOT EXISTS get_animals_by_breed(
    IN animal_breed VARCHAR(255)
)
BEGIN
SELECT * FROM animals WHERE breed = animal_breed;
END |

DELIMITER ;
```

Использование аргументов в процедурах (IN) (вызов процедуры)

Используем новую процедуру **get_animals_by_breed** с аргументом, чтобы получить всех волков:

```
CALL get_animals_by_breed('Wolf');
```

id	name	breed	age	food_kg_per_day
3	Father	Wolf	10	5
4	Raksha	Wolf	11	6
5	Akela	Wolf	15	7

Использование процедур для модификации данных (создание процедуры)

На примере аргумента **IN** покажем, что в числе прочего процедуру можно использовать не только для получения данных из таблицы, но и для вставки, обновления и удаления. Создадим процедуру, которая удаляет из таблицы информацию о животном, а затем при помощи нее удалим слона Хатхи (id = 8):

```
DELIMITER |

CREATE PROCEDURE IF NOT EXISTS delete_animal_by_id(
    IN animal_id INT
)
BEGIN
DELETE FROM animals WHERE id = animal_id;
END |

DELIMITER ;

CALL delete_animal_by_id(8);
```

Использование процедур для модификации данных (результат применения)

Если мы выведем все записи таблицы при помощи **SELECT**, то убедимся, что информация о Хатхи отсутствует:

id	name	breed	age	food_kg_per_day
1	Shere Khan	Tiger	8	12
2	Bagheera	Panther	6	7
3	Father	Wolf	10	5
4	Raksha	Wolf	11	6
5	Akela	Wolf	15	7
6	Baloo	Bear	20	10
7	Kaa	Python	7	2

Использование аргументов в процедурах (OUT) (создание процедуры)

Сейчас же попробуем создать процедуру с переменной типа **OUT**, которую можно использовать после выполнения самой процедуры, но без явного возвращения. Попробуем получить количество денег, которое тратится на среднего представителя какого-либо типа за какой-либо период времени (его мы будем передавать как аргумент в днях).

```
DELIMITER |

CREATE PROCEDURE IF NOT EXISTS get_food_price(
    IN animal_breed VARCHAR(255),
    IN price_per_kg DECIMAL(20,2),
    IN days INT,
    OUT result DECIMAL(20,2)
)
BEGIN

-- переменная для получения среднего количества кг за день
DECLARE medium_amount_of_food_kg_per_day DECIMAL(20,3);

-- получение в переменную среднего количества кг за день
SELECT AVG(food_kg_per_day) INTO medium_amount_of_food_kg_per_day
FROM animals WHERE breed = animal_breed;

-- записываем в переменную типа OUT результат
SELECT medium_amount_of_food_kg_per_day * price_per_kg * days INTO result;

END |

DELIMITER ;
```

Использование аргументов в процедурах (OUT) (вызов процедуры)

Определим с помощью нашей новой процедуры сколько стоит прокорм среднего волка за 30 дней:

```
CALL get_food_price('Wolf', 10, 30, @price_of_food);

SELECT @price_of_food;
```

Заметим, что мы ничего не получили в качестве возвращаемого значения, но смогли получить результат с помощью глобальной переменной **@price_of_food**, которую мы передали в процедуру.

@price_of_food
1800.00

Использование аргументов в процедурах (INOUT) (создание процедуры)

Попробуем реализовать процедуру, где будет использован аргумент **INOUT**. Допустим, мы хотим получить строку о животном по его идентификатору. Кроме того, мы по какой-то причине хотим знать, сколько раз мы вызывали эту процедуру. Для этого после создания процедуры мы объявим глобальную переменную @visits (равную нулю), которую каждый раз будем передавать в процедуру, а внутри процедуры будем увеличивать ее на единицу.

```
DELIMITER |

CREATE PROCEDURE IF NOT EXISTS get_animal_info(
    IN animal_id INT,
    INOUT counter INT
)
BEGIN

SET counter = counter + 1;
SELECT * FROM animals WHERE id = animal_id;

END |

DELIMITER ;
```

Использование аргументов в процедурах (INOUT) (вызов процедуры)

В данном случае не будем обращать внимания на информацию о животных, а сразу перейдем к последней строке, которая получает значение из переменной **@visits**. Это значение будет равно трем, т.к. процедура **get_animal_info** была вызвана три раза.

```
SET @visits = 0;

CALL get_animal_info(5, @visits);
CALL get_animal_info(7, @visits);
CALL get_animal_info(2, @visits);

SELECT @visits;
```

@visits
3

Цикл LOOP

Введение

В MySQL в процедурах (как и в функциях) разрешено использовать т.н. циклы - специальные конструкции, позволяющие повторять определенный блок SQL кода некоторое количество раз. Это может быть полезно, чтобы что-нибудь последовательно посчитать (сложить или умножить несколько раз) или вывести в необходимом нам текстовом виде.

Схема цикла LOOP

Существует три вида циклов, но в данной подглаве мы рассмотрим цикл **LOOP**, который повторяется бесконечно, пока мы его принудительно не остановим. Перед циклом можно поставить специальную метку с любым названием, но после названия следует ставить двоеточие (**:**). Метка нужна для того, чтобы мы смогли остановить цикл, когда выполнится (можно проверять условие внутри цикла с помощью конструкции **IF**, которую мы уже рассматривали в функциях) - для этого стоит использовать конструкцию **LEAVE** имя_начальной_метки. Также можно пропустить итерацию цикла при помощи конструкции - **ITERATE** имя_начальной_метки - в этом случае все инструкции в цикле, которые находятся после команды **ITERATE**, будут пропущены, и цикл повторится опять. Метку можно ставить и после цикла, но в этом нет большого смысла. Завершается объявление цикла ключевой конструкцией **END LOOP**. Все, что находится между **LOOP** и **END LOOP** называется телом цикла - именно тело повторяется то количество раз, которое нам необходимо.

```
[начальная метка:] LOOP

    тело цикла (набор SQL команд);

END LOOP [конечная метка];
```

Создание процедуры с циклом LOOP

Попробуем применить знание о цикле **LOOP** на практике и посчитаем сумму нашего вклада, если известна первоначальная сумма вклада, количество лет, которые вклад будет лежать в банке, а также процент за каждый год, который вклад находился в банке. Важный момент - банковский процент рассчитывается не от первоначального вклада, а от вклада, к которому уже прибавлены проценты от прошедшего года. Таким образом, каждую итерацию сумма, от которой надо высчитывать процент, будет увеличиваться.

```
DELIMITER |

CREATE PROCEDURE IF NOT EXISTS get_final_deposit(
    IN deposit DECIMAL(20,2),
    IN years INT,
    IN percent_per_year DECIMAL(20,2)
)
BEGIN

DECLARE current_year INT DEFAULT 0;
-- объявление цикла и его начальной метки
loop_label: LOOP
    -- если мы уже обошли все годы - прерываем цикл
    IF current_year >= years THEN
        LEAVE loop_label;
    END IF;

    -- рассчитываем депозит на конкретный год
    SET deposit = deposit + (deposit / 100 * percent_per_year);

    -- переходим к следующему году
    SET current_year = current_year + 1;
END LOOP;

SELECT deposit;

END |

DELIMITER ;
```

Применение процедуры с циклом LOOP

Теперь посчитаем сумму вклада, если этот вклад, например, изначально равен 1000 евро, он пролежал в банке 3 года, и каждый год к нему прибавлялось по 5 процентов:

```
CALL get_final_deposit(1000, 3, 5);
```

deposit
1157.63

Цикл WHILE

Схема цикла WHILE

WHILE отличается от **LOOP** тем, что у него есть встроенное условие выхода из цикла (проверяется перед каждой итерацией). Сразу после ключевого слова **WHILE** необходимо разместить конструкцию, которая и проверяет это условие (например, проверка равенства или неравенства между двумя значениями, сравнение чисел и т.д.). После проверки условия надо написать ключевое слово **DO**, затем идет непосредственно тело цикла, а в конце нам надо разместить конструкцию **END WHILE**.

```
[начальная_метка:] WHILE проверка_условия DO

    тело цикла (набор SQL команд);

END WHILE [конечная_метка];
```

WHILE как и **LOOP** поддерживает использование меток. Если мы хотим выйти из цикла вне зависимости от основного условие, то мы можем разместить в цикле наше собственное условие и в тот момент, когда сработает это дополнительное условие, применить конструкцию **LEAVE** имя_начальной_метки. Также мы можем пропустить итерацию при помощи метки **ITERATE** имя_начальной_метки.

Создание процедуры с циклом WHILE

Все циклы позволяют размещать в своем теле другие циклы, что мы продемонстрируем на примере **WHILE**. Попытаемся с помощью вложенных циклов отобразить таблицу умножения произвольного размера - верхний цикл будет отвечать за ряды, а вложенный внутренний цикл - за колонки.

```
DELIMITER |

CREATE PROCEDURE IF NOT EXISTS get_multiplication_table(
   IN max_number INT
)
BEGIN

DECLARE current_row INT DEFAULT 1;
DECLARE current_col INT DEFAULT 1;
DECLARE result TEXT DEFAULT '\n';

WHILE current_row <= max_number DO
   WHILE current_col <= max_number DO
      SET result = CONCAT(result, current_row * current_col, '\t');
      SET current_col = current_col + 1;
   END WHILE;

   SET result = CONCAT(result, '\n');
   SET current_row = current_row + 1;
   SET current_col = 1;
END WHILE;

SELECT result;

END |

DELIMITER ;
```

Применение процедуры с циклом WHILE

Далее отобразим таблицу умножения 8 x 8:

```
CALL get_multiplication_table(8);
```

result							
1	2	3	4	5	6	7	8
2	4	6	8	10	12	14	16
3	6	9	12	15	18	21	24
4	8	12	16	20	24	28	32
5	10	15	20	25	30	35	40
6	12	18	24	30	36	42	48
7	14	21	28	35	42	49	56
8	16	24	32	40	48	56	64

Цикл REPEAT

Схема цикла REPEAT

Цикл **REPEAT** отличается от цикла **WHILE** тем, что условие выхода из цикла проверяется не в самом начале итерации, а в самом конце каждой итерации при помощи ключевого слова **UNTIL** (после которого идет проверка равенства или неравенства между несколькими значениями и т.д.). Таким образом, цикл **WHILE** гарантированно исполнится по крайней мере один раз. Все остальное - объявление меток, выход из цикла при помощи ключевого слова **LEAVE** и пропуск итерации при помощи конструкции **ITERATE** - работает так же как в **LOOP** и **WHILE**. Заканчивается данный тип цикла конструкцией **END REPEAT**.

```
[начальная_метка:] REPEAT

    тело цикла (набор SQL команд);

UNTIL проверка_условия
END REPEAT [конечная_метка];
```

Создание процедуры с циклом REPEAT

Попробуем использовать цикл **REPEAT**, чтобы создать процедуру, подсчитывающуют факториал переданного числа (произведение всех чисел от 1 до переданного числа):

```
DELIMITER |

CREATE PROCEDURE IF NOT EXISTS factorial(
    IN number INT
)
BEGIN

DECLARE result INT DEFAULT 1;
DECLARE current_num INT DEFAULT 1;

REPEAT
    SET result = result * current_num;
    SET current_num = current_num + 1;

UNTIL current_num > number
END REPEAT;

SELECT result;

END |

DELIMITER ;
```

Применение процедуры с циклом REPEAT

Посчитаем с помощью нашей новой процедуры факториал числа 5:

```
CALL factorial(8);
```

result
120

Курсоры

Введение

Во всех предыдущих случаях мы использовали циклы, чтобы сделать повторение какого-то блока кода определенное количество раз и использовали каждую итерацию для внесения определенного вклада в результат процедуры. Однако в некоторых случаях нам надо не просто повторение, а обход строк некой таблицы. То есть, мы хотим на каждой итерации цикла получать новую строку таблицы до того момента, как они кончатся. Теоретически это можно сделать вручную, сначала получив мнимальный идентификатор (а через него и строку таблицы), потом следующий идентификатор после минимального, потом второй, третий и т. д. Однако это слишком громоздко и крайне неоптимально по производительности. MySQL позволяет решить эту проблему при помощи курсоров.

Курсор (**CURSOR**) - это способ получить доступ к каждой конкретной строке таблицы, полученной с помощью запроса. Имея этот доступ, мы можем получить значения столбцов таблицы, вызвать какие-то другие процедуры или функции, что-то посчитать и записать в некий обобщающий результат и только потом перейти в следующей строке. При использовании курсора необходимо помнить о двух его основных особенностях. Во-первых, курсор всегда идет вперед без пропусков строк - нельзя вернутся к предыдущей строке или перепрыгнуть на несколько строк впредет. Во-вторых, курсор можно использовать только для чтения данных, но вставка, обновление и удаление запрещены.

Объявляется курсор с помощью ключевого слова **DECLARE**, после которого располагается название курсора и непосредственно само слово **CURSOR**. После этого ставится ключевое слово **FOR** и следует некий запрос на выборку значений столбцов из таблицы или нескольких таблиц.

```
DECLARE cursor_name CURSOR FOR SELECT column_name FROM ...
```

Создание процедуры с курсором без перехватчика ошибки остановки

Попробуем определить курсор для процедуры, которая получает все имена зверей из таблицы animals и склеивает их в одну строку через пробел (это можно сделать и при помощи агрегатной функции **GROUP_CONCAT**, но для примера сделаем это именно при помощи курсора).

```
DELIMITER |

CREATE PROCEDURE IF NOT EXISTS get_animals_names()
BEGIN

-- объявляем переменную для финального результата
DECLARE result TEXT DEFAULT '';

-- переменная для записи имени для каждой строки
DECLARE animal_name VARCHAR(255) DEFAULT '';

-- создание курсора
DECLARE cur CURSOR FOR SELECT name FROM animals;

-- перед использованием курсор надо всегда открывать
OPEN cur;

animals_loop: LOOP
   -- запись значения столбца в переменную
   FETCH cur INTO animal_name;
   SET result = CONCAT(result, animal_name, ' ');
END LOOP;

-- после использования курсор надо всегда закрывать
CLOSE cur;

SELECT result;
END |

DELIMITER ;
```

Вызов процедуры с курсором без перехватчика ошибки остановки

Процедура, содержащая курсор, будет создана успешно, однако если мы попытаемся ее вызвать, то в ответ получим ошибку. Почему же так? Внимательный читатель сразу заметит, что у цикла **LOOP** нет условия прекращения итераций, т.е. по сути мы должны повторяться бесконечно. Это действительно так, но в данном случае не это главное. Дело в том, что когда у курсора закончатся строки, он вернет ошибку автоматически, даже если мы действительно попытаемся поставить условие выхода из цикла.

```
CALL get_animals_names();
```

Создание процедуры с курсором и перехватчиком ошибки остановки

Для решения проблемы из предыдущего пункта в MySQL есть специальный обработчик, который может “перехватить” ошибку и решить, что делать дальше. Для окончания перебора строк из курсора есть специальная ошибка **NOT FOUND**, поэтому нам не надо смотреть ее код в общей таблице кодов ошибок MySQL.

С помощью обработчика мы реализуем следующую логику - объявим переменную **done**, равную **FALSE**, и при ошибке курсора в обработчике преобразуем ее в **TRUE**. На каждой итерации цикла после попытки получения значений полей из курсора мы будем проверять значение переменной done - если оно равняется **TRUE**, значит курсор вернул ошибку окончания строк и мы можем спокойно выйти из цикла при помощи конструкции **LEAVE**:

```
-- сначала удалим старую неправильную процедуру
DROP PROCEDURE IF EXISTS get_animals_names;

DELIMITER |

CREATE PROCEDURE IF NOT EXISTS get_animals_names()
BEGIN

-- объявляем переменную для финального результата
DECLARE result TEXT DEFAULT '';

-- переменная для записи имени для каждой строки
DECLARE animal_name VARCHAR(255) DEFAULT '';

-- переменная для проверки окончания цикла
DECLARE done INT DEFAULT FALSE;

-- создание курсора
DECLARE cur CURSOR FOR SELECT name FROM animals;

-- создание обработчика ошибок
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

-- перед использованием курсор надо всегда открывать
OPEN cur;

animals_loop: LOOP
   -- запись значения столбца в переменную
   FETCH cur INTO animal_name;

   -- если была ошибка - выходим из цикла
   IF done THEN
      LEAVE animals_loop;
   END IF;

   SET result = CONCAT(result, animal_name, ' ');
END LOOP;

-- после использования курсор надо всегда закрывать
CLOSE cur;

SELECT result;
END |

DELIMITER ;
```

Вызов процедуры с курсором и перехватчиком ошибки остановки

Вот теперь мы можем получить имена всех зверей в виде одной строки без ошибок:

```
CALL get_animals_names();
```

result
Shere Khan Bagheera Father Raksha Akela Baloo Kaa

Обработка иных ошибок

Обратим внимание на то, что иногда могут возникнуть ошибки, отличные от простого завершения строк в курсоре. В этом случае необходимо создать другие обработчики уже для этих ошибок. Кроме того, совершенно необязательно продолжать нормальное исполнение процедуры, вместо ключевого слова **CONTINUE** в обработчике ошибок можно указать **EXIT** - произойдет немедленный выход из того блока, в котором произошла ошибка (в нашем случае произойдет выход из цикла):

```
DECLARE EXIT HANDLER FOR NOT FOUND BEGIN END;
```

Триггеры

Введение

Триггером называется процедура особого типа, которая заранее создается пользователем с целью автоматического вызова при модификации данных во всей таблице или каком-нибудь столбце. Пользователь не может вызывать триггер вручную. В свою очередь под модификацией данных подразумеваются три операции - вставка (**INSERT**), обновление (**UPDATE**) и удаление (**DELETE**).

Создание триггера производится при помощи ключевой конструкции **CREATE TRIGGER** (после чего идет название триггера - может быть дополнено проверкой **IF NOT EXISTS**). Затем определяется момент запуска триггера с помощью ключевого слова **BEFORE** (триггер запускается до выполнения связанного с ним события) или **AFTER** (после события), за которым идет само название операции (**INSERT**, **UPDATE** или **DELETE**). Далее указываем для какой таблицы создается триггер (пишем ключевое слово **ON** и имя таблицы). Далее идет самое интересное - следует написать ключевую конструкцию **FOR EACH ROW**, после которой и будет происходить обработка значений столбцов вставленных, обновленных или модифицированных строк. Закрывается триггер ключевым словом **END**.

```
CREATE
    [DEFINER = user]
    TRIGGER [IF NOT EXISTS] имя_триггера
    { BEFORE | AFTER }
    { INSERT | UPDATE | DELETE }
    ON имя_таблицы
        FOR EACH ROW
            тело_триггера
        END
```

Получить доступ к значениям столбцов модифицируемой строки можно двумя способами. Представим, что у нас в таблице есть столбец name. Если мы хотим узнать его значение перед изменением, то мы можем обратиться к нем как **OLD**.name (используется ключевое слово **OLD**), а если надо узнать его значение после модификации, то используется ключевое слово **NEW** - **NEW**.name. Очевидно, что для модификации типа **INSERT** (вставка) узнать старое значение (**OLD**) невозможно (у нас ничего не было до вставки), также невозможное узнать новое значение (**NEW**) для модификации типа **DELETE** (удаление) (у нас ничего не осталось после вставки).

Особенности триггеров

- Код триггера связан с одной конкретной таблицей и автоматически уничтожается после удаления этой таблицы.

- В случае обнаружения ошибки или нарушения целостности данных все модифицированные триггером данные вернутся в исходное положение. Кроме того, если триггер сработал до модификации данных в таблице (**BEFORE**) и вернул ошибку, то изначальная модификация данных в таблице также будет отменена.

- Основными задачами триггера можно назвать проверку на правильность входящих данных при вставке или обновлении, журналирование (логирование) информации особой важности, а также чистка устаревших и неактуальных данных.

- Всего есть шесть видов триггеров - **BEFORE INSERT**, **AFTER INSERT**, **BEFORE UPDATE**, **AFTER UPDATE**, **BEFORE DELETE**, **AFTER DELETE**

Подготовка (создание таблиц и добавление данных)

```
CREATE TABLE IF NOT EXISTS clients (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE,
    status VARCHAR(255) NOT NULL
);

CREATE TABLE IF NOT EXISTS clients_log (
    id INT UNSIGNED NOT NULL,
    name VARCHAR(255) NOT NULL,
    status VARCHAR(255) NOT NULL,
    action VARCHAR(255) NOT NULL,
    modified_at DATETIME NOT NULL
);

CREATE TABLE IF NOT EXISTS coupons (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    client_id INT UNSIGNED NOT NULL,
    discount DECIMAL(20,2)
);
```

Использование триггеров при вставке данных (BEFORE INSERT / AFTER INSERT)

Введение

Предположим, что наша система, состоящая из трех таблиц из предыдущей главы, имеет очень строгое ограничение - в таблицу ***clients*** могут попасть только те люди, электронная почта которых расположена на серверах Google (т.е. она должна заканчиваться на gmail.com). Проверить это довольно непросто - электронная почта может быть уникальной, может быть меньше или 255 символам, но все равно нам не подходить. Можно, конечно, попробовать проверить почту при помощи ограничения **CHECK**, но условимся, что на этот раз сделаем это при помощи триггера **BEFORE INSERT**.

Как же мы реализуем логику проверки? Ответ очевиден - до вставки данных получим новое значение столбца email, при помощи знакомой нам функции **RIGHT** возьмем последние 10 символов из этого значения, а затем проверим, равняются ли эти символы тексту @gmail.com. Если равняется - мы ничего не делаем, и новая строка благополучно заносится в таблицу ***clients***. Если же не равняется - мы вызовем ошибку MySQL и данные не попадут в таблицу (мы уже говорили об этой особенности триггеров в предыдущей подглаве). Идея выглядит неплохо, только есть одна проблема - мы пока не умеем вручную вызывать ошибку. Что же, пришло время исправить этот недочет.

Создание ошибки производится следующим образом. Сначала надо написать ключевое слово **SIGNAL SQLSTATE**, затем код ошибки - пятизначное число, которое в нашем случае должно начинаться с текста ‘40’ (подробнее о кодах можно почитать на официальном сайте MySQL), а затем может идти необязательная конструкция **SET MESSAGE_TEXT** после которой можно поместить знак = и текстовое разъяснение ошибки в кавычках:

```
SIGNAL SQLSTATE '40000' SET MESSAGE_TEXT = 'You provided a wrong email.';
```

Создание триггера (BEFORE INSERT)

Пришло время создать наш первый триггер, который будет проверять электронную почту на правильность и препятствовать вставке неверных данных:

```
DELIMITER |

CREATE TRIGGER IF NOT EXISTS check_email
BEFORE INSERT
ON clients

FOR EACH ROW BEGIN
   IF RIGHT(NEW.email, 10) <> '@gmail.com' THEN
      SIGNAL SQLSTATE '40000'
      SET MESSAGE_TEXT = 'You provided a wrong email.';
   END IF;
END;
|

DELIMITER ;
```

Обратим внимание, что мы опять применили способ замены разделителя. Как мы видим, внутри триггера также находятся конструкции, которые необходимо завершать точкой с запятой, поэтому если бы мы оставили стандартный разделитель (;), то мы вообще не смогли бы создать триггер.

Применение триггера (BEFORE INSERT) для неправильных данных

Теперь попытаемся вставить в таблицу ***clients*** клиента с неправильной электронной почтой:

```
DECLARE EXIT HANDLER FOR NOT FOUND BEGIN END;
```

Естественно, что в ответ мы получим ошибку '***ERROR 1644 (40000): You provided a wrong email.***', т.к. электронная почта заканчивается запретными символами @inbox.com.

Применение триггера (BEFORE INSERT) для правильных данных

Теперь вставим клиента с правильной электронной почтой и убедимся, что все пройдет успешно:

```
INSERT INTO clients (name, email, status)
VALUES (
   'John Smith', 'john.smith@gmail.com', 'REGULAR_CLIENT'
);
```

id	name	email	status
1	John Smith	john.smith@gmail.com	REGULAR_CLIENT

Создание триггера (AFTER INSERT)

Далее подробно рассмотрим логику работы триггера типа **AFTER INSERT**. Предположим в нашей компании было принято решение каждому клиенту, имеющему статус ***BUSINESS_CLIENT***, при регистрации дарить скидочный купон - -50 процентов на любую покупку. Кажется очевидным, что прописывать создание купона вручную после регистрации не самый лучший вариант, который, в числе прочего, чреват ошибками - можно просто забыть создать купон. Поэтому лучшим вариантом является создание триггера, который после создания клиента сам проверит его статус и при необходимости создаст нужный купон:

```
DELIMITER |

CREATE TRIGGER IF NOT EXISTS generate_business_coupon
AFTER INSERT
ON clients

FOR EACH ROW BEGIN
   IF NEW.status = 'BUSINESS_CLIENT' THEN
      INSERT INTO coupons (
         client_id, discount
      ) VALUES (
         NEW.id, 50
      );
   END IF;
END;
|

DELIMITER ;
```

Применение триггера (AFTER INSERT)

Вставим двух клиентом - одного простого, а второго - со статусом ***BUSINESS_CLIENT***, и посмотрим, как на это отреагирует наш новый триггер:

```
INSERT INTO clients (name, email, status)
VALUES
('Michael Johnson', 'mike.johnson@gmail.com', 'REGULAR_CLIENT'),
('Richard Brown', 'richard.brown@gmail.com', 'BUSINESS_CLIENT');
```

Во-первых, убедимся, что все клиенты были вставлены успешно (таблица ***clients***):

id	name	email	status
1	John Smith	john.smith@gmail.com	REGULAR_CLIENT
2	Michael Johnson	mike.johnson@gmail.com	REGULAR_CLIENT
3	Richard Brown	richard.brown@gmail.com	BUSINESS_CLIENT

Во-вторых, убедимся, что только последний клиент - Richard Brown, имеющий статус BUSINESS_CLIENT и получивший id = 3, автоматически получил свой скидочный купон (в таблице ***coupons***):

id	client_id	discount
1	3	50.00

Использование триггеров при обновлении данных (BEFORE UPDATE / AFTER UPDATE)

Введение

В этом подразделе подробно рассмотрим поведение триггеров при обновление данных (**UPDATE**) в наших таблицах. Предположим, что данные наших клиентов - т.н. информация строгой отчетности, т.е. мы не просто должны видеть в таблице clients актуальную на данный момент информацию, но в случае необходимости мы должны получить всю историю изменений - например, когда и у клиента менялось имя (такое часто происходит при вступлении в брак), как менялся статус клиента и т.д. В этом случае нам на помощь снова спешит триггер, который должен отрабатывать **BEFORE UPDATE** (перед обновлением).

Создание триггера (BEFORE UPDATE)

Внутри триггера мы поступим следующим образом - перед каждым обновлением будем записывать старую (**OLD**) информацию каждого клиента в таблицу ***clients_log***, кроме того, также будем помечать время обновления (получать время будем при помощи встроенной функции **NOW**), а также записывать в столбец action уведомление о том, что это было именно обновление (**UPDATE**):

```
DELIMITER |

CREATE TRIGGER IF NOT EXISTS update_clients_log_on_update
BEFORE UPDATE
ON clients

FOR EACH ROW BEGIN
   INSERT INTO clients_log (
      id, name, status, action, modified_at
   ) VALUES (
      OLD.id, OLD.name, OLD.status, 'UPDATE', NOW()
   );
END;
|

DELIMITER ;
```

Применение триггера (BEFORE UPDATE)

Попытаемся поменять имена и статусы у двух последних клиентов, а потом проверим, как это отразиться на таблице ***clients_log***.

```
UPDATE clients SET name = 'Michael Jordan', status = 'BUSINESS_CLIENT' WHERE id = 2;

UPDATE clients SET name = 'Richard Black', status = 'REGULAR_CLIENT' WHERE id = 3;
```

Таблица ***clients*** должна была спокойно обновиться без каких-либо происшествий:

id	name	email	status
1	John Smith	john.smith@gmail.com	REGULAR_CLIENT
2	Michael Jordan	mike.johnson@gmail.com	BUSINESS_CLIENT
3	Richard Black	richard.brown@gmail.com	REGULAR_CLIENT

В свою очередь в таблице ***clients_log*** должны находиться старые данные из обновленных строк таблицы ***clients***:

id	name	status	action	modified_at
2	Michael Johnson	REGULAR_CLIENT	UPDATE	2023-02-26 19:41:36
3	Richard Brown	BUSINESS_CLIENT	UPDATE	2023-02-26 19:41:36

Создание триггера (AFTER UPDATE)

Рассмотрим работу с триггером после обновления (**AFTER UPDATE**) таблицы и попробуем решить следующую проблему - почему клиенту со статусом **BUSINESS_CLIENT** при создании вручается купон со скидкой, а при обновлении - нет. Создадим триггер, который предоставляет скидку клиенту с бизнес-статусом, но только в том случае, если у клиента еще не было скидки до этого момента:

```
DELIMITER |

CREATE TRIGGER IF NOT EXISTS generate_business_coupon_on_update
AFTER UPDATE
ON clients

FOR EACH ROW BEGIN
   DECLARE number_of_coupons INT DEFAULT 0;
   IF NEW.status = 'BUSINESS_CLIENT' THEN
      SELECT COUNT(*) INTO number_of_coupons
      FROM coupons WHERE client_id = NEW.id;

      IF number_of_coupons = 0 THEN
         INSERT INTO coupons (client_id, discount) VALUES (NEW.id, 50);
      END IF;
   END IF;
END;
|

DELIMITER ;
```

Применение триггера (AFTER UPDATE)

После создания триггера поменяем статус у первого и третьего клиента на **BUSINESS_CLIENT**:

```
UPDATE clients SET status = 'BUSINESS_CLIENT' WHERE id IN (1, 3);
```

id	name	email	status
1	John Smith	john.smith@gmail.com	BUSINESS_CLIENT
2	Michael Jordan	mike.johnson@gmail.com	BUSINESS_CLIENT
3	Richard Black	richard.brown@gmail.com	BUSINESS_CLIENT

А вот если мы обратимся в таблице coupons, то мы увидим, что новый скидочный купон появился только у первого клиента, а у третьего по прежнему остался один купон (у него уже был этот купон до обновления - именно поэтому триггер не стал присваивать ему новый).

id	client_id	discount
1	3	50.00
2	1	50.00

Использование триггеров при удалении данных (BEFORE DELETE / AFTER DELETE)

Создание триггера (BEFORE DELETE)

Мы уже делали запись изменений модификаций данных клиентов с помощью триггера, однако мы не учли один момент. Что будет если мы вставим клиента, а потом удалим без какого-либо обновления? Информация о клиенте пропадет навсегда без какой-либо записи в лог. Поэтому создадим триггер, который записывает в лог информацию о клиенте до удаления клиента (**BEFORE DELETE**), а также помечает время удаления и говорит, что действием (столбец ***action***) - было именно удаление (**DELETE**):

```
DELIMITER |

CREATE TRIGGER IF NOT EXISTS update_clients_log_on_delete
BEFORE DELETE
ON clients

FOR EACH ROW BEGIN
   INSERT INTO clients_log (
      id, name, status, action, modified_at
   ) VALUES (
      OLD.id, OLD.name, OLD.status, 'DELETE', NOW()
   );
END;
|

DELIMITER ;
```

Применение триггера (BEFORE DELETE)

Теперь удалим клиента Michael Jordan (с id = 2):

```
DELETE FROM clients WHERE id = 2;
```

id	name	email	status
1	John Smith	john.smith@gmail.com	BUSINESS_CLIENT
3	Richard Black	richard.brown@gmail.com	BUSINESS_CLIENT

Далее проверим таблицу ***clients_log*** и убедимся, что в самом конце была автоматически вставлена строка с данными удаленного клиента:

id	name	status	action	modified_at
2	Michael Johnson	REGULAR_CLIENT	UPDATE	2023-02-26 19:41:36
3	Richard Brown	BUSINESS_CLIENT	UPDATE	2023-02-26 19:41:36
1	John Smith	REGULAR_CLIENT	UPDATE	2023-02-26 19:41:55
3	Richard Black	REGULAR_CLIENT	UPDATE	2023-02-26 19:41:55
2	Michael Jordan	BUSINESS_CLIENT	DELETE	2023-02-26 19:51:00

Создание триггера (AFTER DELETE)

В конце попытаемся применить триггер типа **AFTER DELETE**. Но сначала обратим внимание на таблицу coupons - в ней есть столбце ***client_id***, который подразумевает ссылку на столбец ***id*** в таблице ***clients***. Однако мы не добавили внешних ключей, чтобы подтвердить эту связь (связь будет отсутствовать, даже если мы добавим внешние ключи, но движком таблиц будет установлен MyISAM). Исправим эту оплошность при помощи триггера, который после удаления клиентов будет явно удалять их купоны.

```
DELIMITER |
CREATE TRIGGER IF NOT EXISTS delete_clients_coupons
AFTER DELETE
ON clients
FOR EACH ROW BEGIN
   DELETE FROM coupons WHERE client_id = OLD.id;
END;
|
DELIMITER ;
```

Применение триггера (AFTER DELETE)

После того, как мы удалим всех клиентов из таблицы ***clients***, то и таблица ***coupons*** будет абсолютно пустой - ее очистит наш новый триггер:

```
DELETE FROM clients;
```

Удаление триггеров

Вручную триггеры удаляются примерно также как функции и процедуры - при помощи команды **DROP TRIGGER**, после которой идет название удаляемого триггера. Если мы не уверены, что триггер существует, и мы хотим избежать потенциальной ошибки, можем применить конструкцию **IF EXISTS**:

```
-- удаление триггера без проверки
DROP TRIGGER delete_clients_coupons;

-- удаление существующего триггера с проверкой
DROP TRIGGER IF EXISTS update_clients_log_on_delete;

-- удаление несуществующего триггера с проверкой
DROP TRIGGER IF EXISTS unexisting_trigger;
```