GeekBrains

Базы данных. Интерактивный курс

Урок 8

Хранимые процедуры и функции, триггеры

Хранимые процедуры и функции

Хранимые процедуры и функции позволяют сохранить последовательность SQL-операторов и вызывать их по имени функции или процедуры:

- CREATE PROCEDURE procedure_name
- CREATE FUNCTION function_name

Разница между процедурой и функцией заключается в том, что функции возвращают значение и их можно встраивать в SQL-запросы, в то время как хранимые процедуры вызываются явно.

Для создания хранимой процедуры предназначен оператор CREATE PROCEDURE, после которого указывается имя процедуры. Давайте создадим процедуру, которая выводит текущую версию MySQL-сервера:

```
CREATE PROCEDURE my_version ()
BEGIN
  SELECT VERSION();
END
```

После команды **CREATE PROCEDURE** указывается имя процедуры и круглые скобки, в которых обычно указывают входящие и исходящие параметры. Мы рассмотрим их чуть позже в рамках текущего урока.

Между ключевыми словами **BEGIN** и **END** размещаются SQL-команды, которые выполняются всякий раз при вызове хранимой процедуры. Итак, нажимаем **ALT + X**, чтобы выполнить запросы.

Чтобы воспользоваться только что созданной хранимой процедурой, используем команду **CALL**, после которой указываем имя вызываемой процедуры:

```
CALL my_version();
```

Мы получили текущую версию MySQL-сервера. Чтобы получить список хранимых процедур, можно воспользоваться командой **SHOW PROCEDURE STATUS**:

```
SHOW PROCEDURE STATUS LIKE 'my_version%'\G
```

Команда возвращает список хранимых процедур и функций. При использовании ключевого слова **LIKE** можно вывести информацию только о тех процедурах, имена которых удовлетворяют шаблону. Для просмотра списка хранимых функций предназначена аналогичная команда **SHOW FUNCTION STATUS**.

Вывод довольно объемный и пользоваться командами **SHOW** не очень удобно, поэтому при наличии прав доступа можно обратиться к системной базе данных mysql, где хранимые процедуры и функции лежат в таблице **proc**.

```
SELECT name, type FROM mysql.proc LIMIT 10;
```

Пользоваться обычным SELECT-запросом гораздо удобнее, можно извлекать только ту информацию, которая действительно необходима. После того, как хранимая процедура уже создана, посмотреть ее содержимое можно при помощи команды **SHOW CREATE PROCEDURE**:

```
SHOW CREATE PROCEDURE my_version\G
```

Для удаления хранимых процедур и функций предназначены операторы **DROP PROCEDURE** и **DROP FUNCTION**. Давайте удалим процедуру **my_version**:

```
DROP PROCEDURE my_version;
```

Попытка удаления несуществующей хранимой процедуры вызывает ошибку:

```
DROP PROCEDURE my_version;
```

Синтаксис команды допускает использование ключевого слова **IF EXISTS**:

```
DROP PROCEDURE IF EXISTS my_version;
```

В этом случае, если хранимой процедуры уже не существует, команда завершается без сообщения об ошибке.

Создание и использование хранимой функции немного отличается от создания и использования хранимой процедуры. Давайте сразу разместим команду удаления хранимой функции, чтобы мы могли многократно вызывать файл **func.sql**:

```
CREATE FUNCTION get_version ()
RETURNS TEXT DETERMINISTIC
BEGIN
  RETURN VERSION();
END
```

Функция создается командой **CREATE FUNCTION**, после которой идет имя функции. Хранимая функция встраивается в SQL-запросы, как обычная mysql-функция. Она должна возвращать значение. Ключевое слово **RETURNS** указывает возвращаемый тип, например **TEXT** мы можем заменить на **VARCHAR(255)**. Ключевое слово **DETERMINISTIC** (дэтеминистик) сообщает, что результат функции детерминирован, т.е., при каждом вызове будет возвращаться одно и то же значение, и если его закешировать в рамках запроса, ничего страшного не произойдет. Если значения, которые возвращает функция, каждый раз различны, то перед **DETERMINISTIC** (дэтеминистик) следует добавить отрицание **NOT**. Далее следует тело функции, которое размещается между ключевыми словами **BEGIN** и **END**. Внутри тела обязательно должно присутствовать ключевое слово **RETURN**, которое возвращает результат вычисления. В данном случае мы просто возвращаем результат вызова mysql-функции **VERSION()**.

Для вызова хранимой функции не требуется специальной команды, как в случае хранимых процедур. Порядок их вызова совпадает с порядком вызова встроенных функций MySQL:

```
SELECT get_version();
```

Основная трудность, которая возникает при работе с хранимыми процедурами и функциями, заключается в том, что символ точки с запятой (;) используется в теле запроса для разделения SQL-команд.

Создание хранимой процедуры или функции — это тоже команда, которая тоже должна завершаться точкой с запятой. В результате возникает конфликт.

```
DROP FUNCTION IF EXISTS get_version;
CREATE FUNCTION get_version ()
RETURNS TEXT DETERMINISTIC
BEGIN
  RETURN VERSION();
END
```

Чтобы его избежать, во всех клиентах предусмотрена возможность переназначать признак окончания запроса, в консольном клиенте **mysql** это осуществляется при помощи команды **DELIMITER**.

```
DELIMITER //
SELECT VERSION()//
DELIMITER ;
SELECT VERSION();
```

Давайте снова назначим разделителем два слеша:

```
DELIMITER //
```

Теперь мы можем воспользоваться новым разделителем

```
CREATE FUNCTION get_version ()
RETURNS TEXT DETERMINISTIC
BEGIN
  RETURN VERSION();
END//
```

Параметры процедур и функций

Хранимые процедуры и функции могут использовать параметры. Параметры могут передавать значения внутрь функции и извлекать результаты вычисления. Для этого каждый из параметров снабжается одним из атрибутов: **IN**, **OUT** или **INOUT**.

Параметр **param_name** предваряет одно из ключевых слов **IN**, **OUT**, **INOUT**, которые позволяют задать направление передачи данных:

- **IN** — данные передаются строго внутрь хранимой процедуры, но если параметру с данным модификатором внутри функции присваивается новое значение, по выходу из нее оно не сохраняется и параметр принимает значение, которое он имел до вызова процедуры.
- **OUT** — данные передаются строго из хранимой процедуры. Даже если параметр имеет какое-то начальное значение, внутри хранимой процедуры оно не принимается во внимание.

С другой стороны, если параметр изменяется внутри процедуры, после ее вызова он имеет значение, присвоенное ему внутри процедуры.

- **INOUT** — значение этого параметра как принимается во внимание внутри процедуры, так и сохраняет свое значение по выходу из нее.

Атрибуты **IN**, **OUT** и **INOUT** доступны лишь для хранимой процедуры, в хранимой функции все параметры всегда имеют атрибут **IN**.

```
DELIMITER //
```

Давайте сразу установим в качестве разделителя два слеша, для этого используем команду **DELIMITER**. Создадим простейшую процедуру **get_x**, которая принимает единственный параметр **value** и устанавливает переменную.

```
CREATE PROCEDURE set_x (IN value INT)
BEGIN
  SET @x = value;
END//
```

Использование ключевого слова **IN** не обязательно — если ни один из атрибут не указан, СУБД MySQL считает, что параметр объявлен с атрибутом **IN**.

В теле процедуры мы используем команду **SET**, чтобы создать пользовательскую переменную **@x**. Напоминаю, что наши собственные переменные создаются с использованием символа **@**.

При вызове хранимой процедуры мы можем передать в круглых скобках значение, которое будет использоваться вместо параметра внутри хранимой функции.

```
CALL set_x(123456)//
```

Такое значение называется аргументом функции. Результатом работы процедуры будет установленная переменная **@x**:

```
SELECT @x//
```

В отличие от пользовательской переменной **@x**, которая является глобальной и доступна как внутри хранимой процедуры **set_x()**, так и вне ее, параметры функции локальны и доступны для использования только внутри функции.

```
DROP PROCEDURE IF EXISTS set_x//
CREATE PROCEDURE set_x (IN value INT)
BEGIN
  SET @x = value;
  SET value = value - 1000;
END//
SET @y = 10000//
CALL set_x(@y)//
SELECT @x, @y//
```

Хранимая процедура **set_x()** принимает единственный IN-параметр **value**, при помощи оператора **SET** его значение изменяется внутри функции. Однако после выполнения хранимой процедуры значение пользовательской переменной **@y**, переданной функции в качестве параметра, не изменяется.

Если требуется, чтобы значение переменной менялось, необходимо объявить параметр процедуры с модификатором **OUT**.

```
DROP PROCEDURE IF EXISTS set_x//
CREATE PROCEDURE set_x (OUT value INT)
BEGIN
  SET @x = value;
  SET value = 1000;
END//
SET @y = 10000//
CALL set_x(@y)//
SELECT @x, @y//
```

При использовании модификатора **OUT** любые изменения параметра внутри процедуры отражаются на аргументе. Передача в качестве значения пользовательской переменной позволяет использовать результат процедуры для дальнейших вычислений. Однако передать значение внутрь функции при помощи OUT-параметра уже не получится.

Чтобы через параметр можно было и передать значение внутрь процедуры, и получить значение, которое попадает в параметр в результате вычислений внутри процедуры, его следует объявить с атрибутом **INOUT**:

```
DROP PROCEDURE IF EXISTS set_x//
CREATE PROCEDURE set_x (INOUT value INT)
BEGIN
  SET @x = value;
  SET value = value - 1000;
END//
SET @y = 10000//
CALL set_x(@y)//
SELECT @x, @y//
```

До этого момента локальные переменные в хранимой процедуре или функции объявлялись как входящие или исходящие параметры, однако это не всегда удобно.

Часто требуется локальная переменная без необходимости передавать или возвращать с ее помощью какие-либо значения:

```
CREATE PROCEDURE declare_var ()
BEGIN
  DECLARE id, num INT(11) DEFAULT 0;
  DECLARE name, hello, temp TINYTEXT;
END//
```

Объявить такую переменную можно при помощи команды **DECLARE**. Один оператор **DECLARE** позволяет объявить сразу несколько переменных одного типа, причем необязательное слово **DEFAULT** позволяет назначить инициирующее значение. Те переменные, для которых не указывается ключевое слово **DEFAULT**, можно инициировать при помощи команды **SET.** Позже в

ролике мы остановимся на этом подробнее. Команда **DECLARE** может появляться только внутри блока **BEGIN...END**, область видимости объявленной переменной также ограничена этим блоком.

```
DROP PROCEDURE IF EXISTS declare_var//
CREATE PROCEDURE declare_var ()
BEGIN
  DECLARE var TINYTEXT DEFAULT 'внешняя переменная';
  BEGIN
      DECLARE var TINYTEXT DEFAULT 'внутренняя переменная';
      SELECT var;
  END;
  SELECT var;
END//
CALL declare_var()//
```

Это означает, что в разных блоках **BEGIN...END** могут быть объявлены переменные с одинаковым именем, и действовать они будут только в рамках одного блока, не пересекаясь с переменными других.

```
CREATE PROCEDURE one_declare_var ()
BEGIN
  DECLARE var TINYTEXT DEFAULT 'внешняя переменная';
  BEGIN
      SELECT var;
  END;
  SELECT var;
END//
```

Однако переменная, объявленная во внешнем блоке **BEGIN...END**, будет доступна во вложенном блоке, если не будет объявлено экранирующей ее переменной.

```
CALL one_declare_var()//
```

Помимо ключевого слова **DEFAULT**, позволяющего присваивать значение переменной при ее объявлении допускается присвоение значения переменной по мере работы процедуры или функции. Существует два способа назначить переменной новое значение:

- Команда **SET**.
- Команда **SELECT ... INTO ... FROM**.

Ниже приводится типичный способ использования **SET**: переменной **var** сначала присваивается значение 100, а второй оператор **SET** увеличивает значение переменной **var** на единицу.

```
SET var = 100;
SET var = var + 1;
```

Команда **SELECT … INTO … FROM** позволяет сохранять результаты SELECT-запроса без их немедленного вывода и без использования внешних переменных.

```
SELECT id, data INTO x, y FROM test LIMIT 1;
```

Тут в примере значения полей **id** и **data** из таблицы **test** присваивается локальным переменным **x** и **y**.

Давайте создадим функцию, которая принимает в качестве аргумента количество секунд и возвращает строку, в которой сообщается, сколько дней, часов, минут и секунд входит в интервал.

```
DROP FUNCTION IF EXISTS second_format;
CREATE FUNCTION second_format (seconds INT)
RETURNS VARCHAR(255) DETERMINISTIC
BEGIN
  RETURN '';
END
```

Давайте начнем с такой заготовки:

```
DROP FUNCTION IF EXISTS second_format;
CREATE FUNCTION second_format (seconds INT)
RETURNS VARCHAR(255) DETERMINISTIC
BEGIN
  DECLARE days, hours, minutes INT;
  RETURN '';
END
```

Объявляем дни, часы и минуты:

```
DROP FUNCTION IF EXISTS second_format;
CREATE FUNCTION second_format (seconds INT)
RETURNS VARCHAR(255) DETERMINISTIC
BEGIN
  DECLARE days, hours, minutes INT;

  SET days = FLOOR(seconds / 86400);
  SET seconds = seconds - days * 86400;
  SET hours = FLOOR(seconds / 3600);
  SET seconds = seconds - hours * 3600;
  SET minutes = FLOOR(seconds / 60);
  SET seconds = seconds - minutes * 60;

  RETURN CONCAT(days, " days ",
            hours, " hours ",
            minutes, " minutes ",
            seconds, " seconds");
END
```

И используем mysql-функцию **CONCAT()**, чтобы соединить результаты в строки:

```
SELECT second_format(123456)//
```

Итак, мы познакомились с командой **SET**, теперь давайте более подробно поговорим о команде **SELECT** и особенностях ее использования внутри хранимых процедур и функций.

```
DROP PROCEDURE IF EXISTS numcatalogs//
CREATE PROCEDURE numcatalogs (OUT total INT)
BEGIN
  SELECT COUNT(*) INTO total FROM catalogs;
END//
```

Давайте создадим процедуру **numcatalogs**, которая возвращает количество строк в таблице **catalogs**. Хранимая процедура **numcatalogs()** имеет один целочисленный (**INT**) параметр **total**, в который сохраняется количество записей в таблице **catalogs**.

Осуществляется это при помощи команды **SELECT** и дополнительного ключевого слова **INTO**, который позволяет сохранять результаты непосредственно в выходном параметре **total** функции **numcatalogs**.

```
CALL numcatalogs(@a)//
SELECT @a//
```

Если команда **SELECT** возвращает несколько значений, их можно принять в несколько переменных; для этого их достаточно перечислить через запятую.

Ветвление

Оператор **IF** позволяет реализовать ветвление программы по условию. **IF** принимает значение либо **TRUE** (истину), либо **FALSE** (ложь). В MySQL **TRUE** и **FALSE** — константы для целочисленных значений 1 и 0. Если логическое выражение истинно, **IF** выполняет SQL-выражения, которые размещаются в теле команды между ключевыми словами **THEN** и **END IF**.

```
DROP PROCEDURE IF EXISTS format_now//
CREATE PROCEDURE format_now (format CHAR(4))
BEGIN
  IF(format = 'date') THEN
      SELECT DATE_FORMAT(NOW(), "%d.%m.%Y") AS format_now;
  END IF;
  IF(format = 'time') THEN
      SELECT DATE_FORMAT(NOW(), "%H:%i:%s") AS format_now;
  END IF;
END//
```

Давайте вызовем процедуру:

```
CALL format_now('date')//
CALL format_now('time')//
```

Команда **IF** поддерживает ключевое слово **ELSE**. Давайте перепишем процедуру **format_now** с использованием **ELSE**:

```
DROP PROCEDURE IF EXISTS format_now//
CREATE PROCEDURE format_now (format CHAR(4))
BEGIN
  IF(format = 'date') THEN
      SELECT DATE_FORMAT(NOW(), "%d.%m.%Y") AS format_now;
  ELSE
      SELECT DATE_FORMAT(NOW(), "%H:%i:%s") AS format_now;
  END IF;
END//
```

Если параметр **format** равен **'date'**, условие в **IF** является истинным, выполняется первый блок, выводящий текущую дату. Если аргумент принимает любое другое условие, условие в **IF** является ложным и запрос выполняется в блоке **ELSE**.

Оператор **IF** позволяет выбрать и большее число альтернатив. Для этого используются дополнительные условия после ключевого слова **ELSEIF**:

```
DROP PROCEDURE IF EXISTS format_now//
CREATE PROCEDURE format_now (format CHAR(4))
BEGIN
  IF(format = 'date') THEN
      SELECT DATE_FORMAT(NOW(), "%d.%m.%Y") AS format_now;
  ELSEIF (format = 'time') THEN
      SELECT DATE_FORMAT(NOW(), "%H:%i:%s") AS format_now;
  ELSE
      SELECT UNIX_TIMESTAMP(NOW()) AS format_now;
  END IF;
END//
```

Процедуру **format_now()** можно изменить таким образом, чтобы она выводила количество секунд, прошедших с полуночи первого января 1970 года. Таким образом, аргумент **format** по-прежнему может принимать значения **'date'** и **'time'** для вывода даты и времени в строковом представлении. Если задан любой другой аргумент — выводится количество секунд, прошедших с полуночи первого января 1970 года:

```
CALL format_now('secs')//
```

Под множественный выбор в MySQL предназначен оператор **CASE**:

```
DROP PROCEDURE IF EXISTS format_now//
CREATE PROCEDURE format_now (format CHAR(4))
BEGIN
  CASE format
      WHEN 'date' THEN
      SELECT DATE_FORMAT(NOW(), "%d.%m.%Y") AS format_now;
      WHEN 'time' THEN
      SELECT DATE_FORMAT(NOW(), "%H:%i:%s") AS format_now;
      WHEN 'secs' THEN
      SELECT UNIX_TIMESTAMP(NOW()) AS format_now;
      ELSE
```

```
      SELECT 'Ошибка в параметре format';
  END CASE;
END//

CALL format_now ('date')//
CALL format_now ('secs')//
CALL format_now ('four')//
```

Циклы

Циклы являются важнейшей конструкцией, без которой хранимые процедуры и функции не имели бы достаточно функциональности. Таблицы, как правило, имеют множество записей, поэтому циклическая обработка данных встречается в SQL-программировании достаточно часто.

MySQL предоставляет три цикла: **while**, **repeat** и **loop.** Их можно использовать в теле хранимой процедуры или функции, т.е., между ключевыми словами **BEGIN** и **END**.

```
CREATE PROCEDURE NOW3 ()
BEGIN
  DECLARE i INT DEFAULT 3;
  WHILE i > 0 DO
      SELECT NOW();
      SET i = i - 1;
  END WHILE;
END//
```

Здесь в процедуре **NOW3** используется цикл **WHILE** для трехкратного вывода даты и времени. Цикл начинается с ключевого слова **WHILE**, после которого следует условие. Условие вычисляется на каждой итерации цикла: если оно возвращает истину (**TRUE**), очередная итерация выполняется, если при очередной проверке оно будет ложным (**FALSE**), цикл завершит работу.

Чтобы не создать бесконечный цикл, условие подбирается таким образом, чтобы рано или поздно оно становилось ложным и цикл прекращал свою работу. Цикл **while**, в свою очередь, сам имеет тело, начало которого обозначается ключевым словом **DO**, а завершение — ключевым словом **END WHILE**.

Все команды, которые располагаются между этими ключевыми словами, выполняются на каждой итерации цикла. Обратите внимание: перед циклом мы заводим переменную **i**, которой при помощи ключевого слова **DEFAULT** устанавливаем значение 3.

На каждой итерации мы уменьшаем значение **i** на единицу: пока **i** больше нуля, условие цикла остается истинным. Как только значение уменьшается до 0, условие возвращает **FALSE** и цикл завершает работу. Таким образом, текущая дата будет выведена только три раза. Давайте в этом убедимся.

```
CALL NOW3()//
```

Количество повторов не обязательно задавать внутри хранимой процедуры. Например, мы можем задать его в качестве входящего параметра.

```
CREATE PROCEDURE NOWN (IN num INT)
BEGIN
  DECLARE i INT DEFAULT 0;
  IF (num > 0) THEN
      WHILE i < num DO
      SELECT NOW();
      SET i = i + 1;
      END WHILE;
  ELSE
      SELECT 'Ошибочное значение параметра';
  END IF;
END//
```

При помощи оператора **IF** можем убедиться, что заданное значение больше 0 и использовать его в условии оператора **while**. Обратите внимание: на этот раз локальная переменная **i** пробегает значение от 0 до заданного в параметре **num**. Как только оно достигает заданного пользователем значения, условие становится ложным и цикл прекращает работу.

Давайте запустим процедуру на выполнение:

```
CALL NOWN(2)//
```

Итак, у нас выводится только две даты. Для досрочного выхода из цикла предназначен оператор **LEAVE**. Давайте ограничим цикл в процедуре **NOWN** только двумя итерациями, т.е., сколько бы выводов пользователь ни заказывал, максимальное количество, которое будет доступно — 2.

```
DROP PROCEDURE IF EXISTS NOWN//
CREATE PROCEDURE NOWN (IN num INT)
BEGIN
  DECLARE i INT DEFAULT 0;
  IF (num > 0) THEN
      cycle: WHILE i < num DO
      IF i >= 2 THEN LEAVE cycle;
      END IF;
      SELECT NOW();
      SET i = i + 1;
      END WHILE cycle;
  ELSE
      SELECT 'Ошибочное значение параметра';
  END IF;
END//
```

В тело цикла добавляется дополнительное if-условие, не допускающее достижение счетчика **i** значения 2. Как только условие срабатывает, выполняется команда **LEAVE**. Циклы можно вкладывать друг в друга, поэтому, чтобы команда **LEAVE** понимала, какой из циклов следует останавливать, ей всегда передается метка цикла, в данном случае **cycle**. Эту метку мы должны поместить перед ключевым словом **WHILE** и после ключевого слова **END WHILE**.

Давайте запросим заведомо огромное значение, например 1000:

```
CALL NOWN(1000)//
```

Как видим, выводятся только две даты, у нас сработал досрочный выход из цикла.

Еще один оператор, выполняющий досрочное прекращение цикла — **ITERATE**. В отличие от оператора **LEAVE**, **ITERATE** не прекращает выполнение цикла, он лишь досрочно прекращает текущую итерацию.

Давайте создадим хранимую процедуру, которая продемонстрирует **ITERATE** на практике:

```
DROP PROCEDURE IF EXISTS numbers_string//
CREATE PROCEDURE numbers_string (IN num INT)
BEGIN
  DECLARE i INT DEFAULT 0;
  DECLARE bin TINYTEXT DEFAULT '';
  IF (num > 0) THEN
      cycle : WHILE i < num DO
      SET i = i + 1;
      SET bin = CONCAT(bin, i);
      IF i > CEILING(num / 2) THEN ITERATE cycle;
      END IF;
      SET bin = CONCAT(bin, i);
      END WHILE cycle;
      SELECT bin;
  ELSE
      SELECT 'Ошибочное значение параметра';
  END IF;
END//

CALL numbers_string(9)//
```

Внутри цикла счетчик **i** пробегает значения от 1 до 9, на каждой итерации значение счетчика добавляется к строке **bin**. Если if-условие ложное, то значение добавляется два раза, если истинное, срабатывает оператор **ITERATE** и текущая итерация завершается досрочно. Именно поэтому в результатах мы видим удвоенные цифры до 5 и одиночные цифры после 5.

Оператор **REPEAT** похож на оператор **WHILE**.

```
DROP PROCEDURE IF EXISTS NOW3//
CREATE PROCEDURE NOW3 ()
BEGIN
  DECLARE i INT DEFAULT 3;
  REPEAT
      SELECT NOW();
      SET i = i - 1;
  UNTIL i <= 0
  END REPEAT;
END//
```

Однако условие для покидания цикла располагается не в начале тела цикла, а в конце. В результате тело цикла в любом случае выполняется хотя бы один раз. В конце цикла после ключевого слова

UNTIL располагается условие; если оно истинно, работа цикла прекращается, если ложно, происходит еще одна итерация.

Эта хранимая процедура должна выполняться в теле цикла три раза. Давайте в этом убедимся.

```
CALL NOW3()//
```

Цикл **LOOP**, в отличие от операторов **WHILE** и **REPEAT**, не имеет условий выхода. Поэтому он должен обязательно иметь в составе оператор **LEAVE**.

```
DROP PROCEDURE IF EXISTS NOW3//
CREATE PROCEDURE NOW3 ()
BEGIN
  DECLARE i INT DEFAULT 3;
  cycle: LOOP
      SELECT NOW();
      SET i = i - 1;
      IF i <= 0 THEN LEAVE cycle;
      END IF;
  END LOOP cycle;
END//
```

Так как мы используем оператор **LEAVE**, мы должны разместить перед ключевым словом **LOOP** и после **END LOOP** метку. Здесь она называется **cycle**.

Запускаем процедуру на выполнение.

```
CALL NOW3()//
```

Обработка ошибок

Во время выполнения хранимых процедур и функций могут происходить самые разнообразные ошибки. Поэтому СУБД MySQL поддерживает обработчики ошибок, которые позволяют каждой ошибке назначить свой собственный обработчик.

Кроме того, обработчик, в зависимости от ситуации и серьезности ошибки, может как прекратить, так и продолжить выполнение процедуры.

```
SHOW CREATE TABLE catalogs\G
```

Давайте смоделируем ошибку. Так как поле **id** объявлено первичным ключом, его значения обязаны быть строго уникальными. Добавление в таблицу значений, совпадающих с одним из тех, которые в ней уже есть, приведет к возникновению ошибочной ситуации.

```
SELECT * FROM catalogs//
INSERT INTO catalogs VALUES (1, 'Процессоры')//
```

Номер 23000 — код ошибки, возникающей при попытке вставить уже существующее значение в уникальный столбец. Мы можем обработать код ошибки при помощи команды **DECLARE ... HANDLER FOR**. Эта команда может появляться только в теле хранимых функций и процедур.

```
DROP PROCEDURE IF EXISTS insert_to_catalog//
CREATE PROCEDURE insert_to_catalog (IN id INT, IN name VARCHAR(255))
BEGIN
   DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @error = 'Ошибка вставки
значения';
  INSERT INTO catalogs VALUES(id, name);
  IF @error IS NOT NULL THEN
      SELECT @error;
  END IF;
END//

SELECT * FROM catalogs//

CALL insert_to_catalog(4, 'Оперативная память')//
CALL insert_to_catalog(1, 'Процессоры')//
```

Если результирующий запрос возвращает одну запись, поместить результаты в промежуточные переменные можно при помощи оператора **SELECT...INTO...FROM**. Однако результирующие таблицы чаще содержат несколько записей, и использование такого запроса совместно с оператором **SELECT...INTO...FROM** приводит к возникновению ошибки:

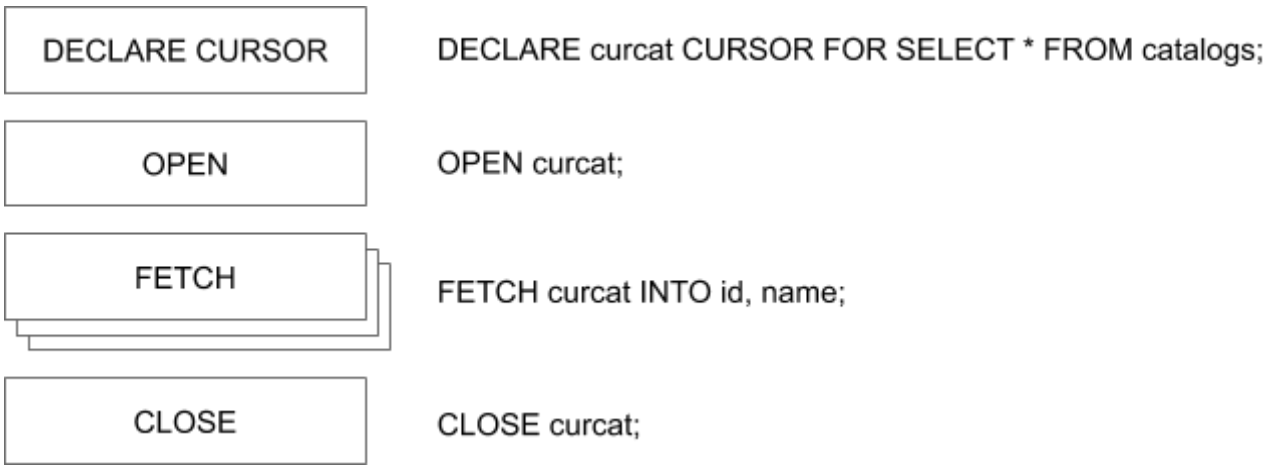
```
DROP PROCEDURE IF EXISTS catalog_id//
CREATE PROCEDURE catalog_id (OUT total INT)
BEGIN
  SELECT id INTO total FROM catalogs;
END//

CALL catalog_id(@total)//
```

Избежать возникновения ошибки можно, добавив **LIMIT 1** или назначив обработчик ошибок. Однако функция будет реализовывать совсем не то поведение, которое ожидает пользователь. Чаще всего требуется обработать именно многострочную результирующую таблицу.

Курсоры

Решить эту задачу можно при помощи курсоров, которые представляют собой своеобразные циклы, специально предназначенные для обхода результирующих таблиц.



Работа с курсорами похожа на работу с файлами — сначала происходит открытие курсора, затем чтение и после закрытие.

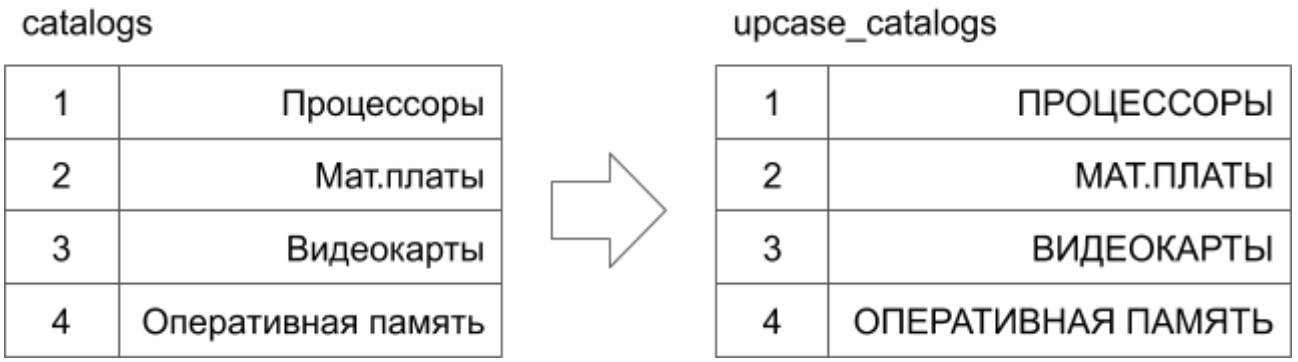
1. При помощи инструкции **DECLARE CURSOR** имя курсора связывается с выполняемым запросом.

2. Оператор **OPEN** выполняет запрос, связанный с курсором, и устанавливает курсор перед первой записью результирующей таблицы.

3. Оператор **FETCH** помещает курсор на первую запись результирующей таблицы и извлекает данные из записи в локальные переменные хранимой процедуры. Повторный вызов оператора **FETCH** приводит к перемещению курсора к следующей записи, и так до тех пор, пока записи в результирующей таблице не будут исчерпаны. Эту операцию удобно осуществлять в цикле.

4. Оператор **CLOSE** прекращает доступ к результирующей таблице и ликвидирует связь между курсором и результирующей таблицей.

Давайте в качестве примера создадим копию таблицы **catalogs** учебной базы данных **shop**. Эту дублирующую таблицу назовем **upcase_catalogs** и поместим в нее записи из оригинальной таблицы, только приведем названия разделов к верхнему регистру.



Мы не знаем заранее, сколько записей может быть в таблице **catalogs**, поэтому проще всего обойти ее при помощи курсора.

Давайте создадим таблицу.

```
DROP TABLE IF EXISTS upcase_catalogs//
CREATE TABLE upcase_catalogs (
  id SERIAL PRIMARY KEY,
```

```
  name VARCHAR(255) COMMENT 'Название раздела'
) COMMENT = 'Разделы интернет-магазина'//

DROP PROCEDURE IF EXISTS copy_catalogs//
CREATE PROCEDURE copy_catalogs ()
BEGIN
  DECLARE id INT;
  DECLARE is_end INT DEFAULT 0;
  DECLARE name TINYTEXT;

  DECLARE curcat CURSOR FOR SELECT * FROM catalogs;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET is_end = 1;

  OPEN curcat;

  cycle : LOOP
      FETCH curcat INTO id, name;
      IF is_end THEN LEAVE cycle;
      END IF;
      INSERT INTO upcase_catalogs VALUES(id, UPPER(name));
  END LOOP cycle;

  CLOSE curcat;
END//

Внутри хранимой процедуры мы
/* Объявляем локальные переменные */
/* Объявляем курсор */
/* Объявляем обработчик для ситуации, когда курсор достигает
      конца результирующей таблицы */
/* Открываем курсор при помощи ключевого слова OPEN */
/*  В цикле читаем данные из курсора и формируем запись для таблицы
upcase_catalogs */
/* В конце закрываем курсор при помощи команды CLOSE */

CALL copy_catalogs()//
SELECT * FROM upcase_catalogs//
```

Как хранимая процедура заполняет таблицу **upcase_catalogs** записями из таблицы **catalogs**, переводя названия разделов в верхний регистр.

Триггеры

Триггер — специальная хранимая процедура, привязанная к событию изменения содержимого таблицы.

Существуют три события изменения таблицы, к которым можно привязать триггер: это изменение содержимого таблицы при помощи команд **INSERT**, **DELETE** и **UPDATE**. Триггеры могут выполняться до и после каждой из этих команд, поэтому существуют три BEFORE- и три AFTER-триггера, которые на рисунке обозначены белыми прямоугольниками.


```
CREATE TRIGGER catalogs_count AFTER INSERT ON catalogs
FOR EACH ROW
BEGIN
  SELECT COUNT(*) INTO @total FROM catalogs;
END//
```

Для создания триггера используется команда **CREATE TRIGGER**. После команды следует имя триггера, далее при помощи ключевого слова **AFTER** указывается, что триггер запускается уже после выполнения команды. В данном случае — после команды **INSERT** для таблицы **catalogs**.

Между ключевыми словами **BEGIN** и **END** располагается тело триггера. Внутри составного тела триггера между ключевыми словами **BEGIN** и **END** допускаются все специфичные для хранимых процедур операторы и конструкции.

В триггере мы извлекаем количество записей в таблице **catalogs** и помещаем это значение в переменную **@total**. Воспользуемся триггером. Для этого достаточно вставить новую запись в таблицу catalogs:

```
INSERT INTO catalogs VALUES (NULL, 'Мониторы')//
```

Извлечем записи:

```
SELECT * FROM catalogs;
```

И давайте убедимся, что переменная **@total** установлена:

```
SELECT @total//
```

Получить список триггеров можно при помощи команды **SHOW TRIGGERS**:

```
SHOW TRIGGERS\G
```

За удаление отвечает команда **DROP TRIGGER**. Давайте удалим ранее созданный триггер **catalogs_count**:

```
DROP TRIGGER catalogs_count//
```

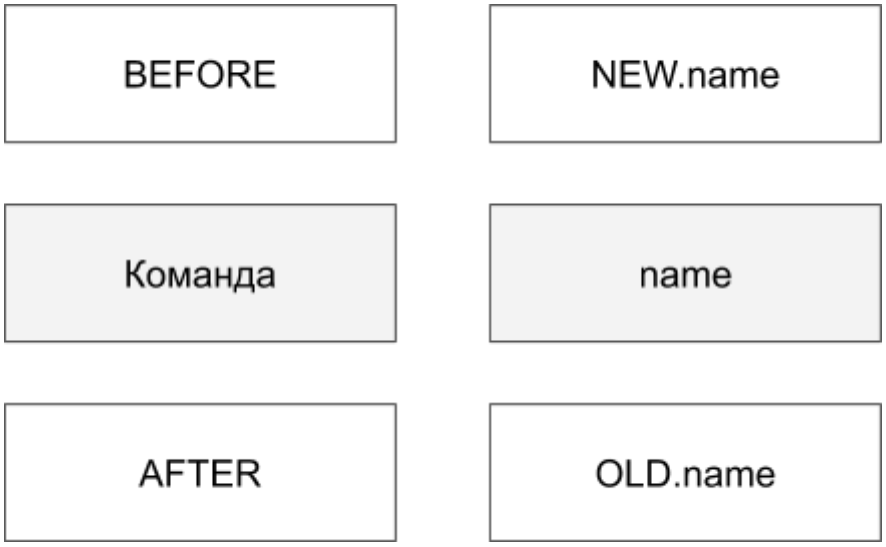
Попытка удаления несуществующего триггера завершается неудачей:

```
DROP TRIGGER catalogs_count//
```

Чтобы избежать ошибки, как и во многих других командах MySQL, мы можем использовать ключевое слово **IF EXISTS**.

```
DROP TRIGGER IF EXISTS catalogs_count//
```

Триггеры очень сложно использовать, не имея доступа к новым записям, которые вставляются в таблицу, или старым записям, которые обновляются или удаляются. Для доступа к новым и старым записям используются префиксы **NEW** и **OLD** соответственно.



То есть, если в таблице обновляется поле **name**, то получить доступ к старому значению можно по имени **OLD.name**, а к новому — **NEW.name**.

Давайте создадим триггер, который при вставке новой товарной позиции в таблицу **products** будет следить за состоянием внешнего ключа **catalog_id.** Если внешний ключ будет оставаться незаполненным, триггер будет извлекать из таблицы **catalogs** наименьший идентификатор **id** и назначать его записи.

Эти действия нужно выполнить до вставки записи в таблицу, поэтому воспользуемся BEFORE-триггером:

```
CREATE TRIGGER check_catalog_id_insert BEFORE INSERT ON products
FOR EACH ROW
BEGIN
  DECLARE cat_id INT;
  SELECT id INTO cat_id FROM catalogs ORDER BY id LIMIT 1;
  SET NEW.catalog_id = COALESCE(NEW.catalog_id, cat_id);
END//
```

В триггере мы объявляем переменную **id** и извлекаем в нее наименьшее значение идентификатора из

таблицы **catalogs**. Далее, если вставляемое значение **catalog_id** не инициализировано, вместо него вставляется значение переменной **id**. Если пользователь передает значение **catalog_id**, оно остается неизменным. Для доступа к данным, которые пользователь хочет вставить в таблицу **products**, мы используем ключевое слово **NEW**.

Функция **COALESCE** возвращает первое не NULL-значение и довольно интенсивно используется в SQL-программировании:

```
SELECT COALESCE(NULL, NULL, NULL, 1, 2, 3)//
SELECT COALESCE(NULL, 3, NULL)//
```

Давайте вставим в таблицу **products** записи без указания внешнего ключа **catalog_id**:

```
INSERT INTO products
  (name, description, price)
VALUES
  ('AMD RYZEN 5 1600', 'Процессор AMD', 13200.00)//

SELECT id, name, price, catalog_id FROM products//
```

Как видим, товарная позиция автоматически получает значение 1, в то же время, если мы вставим внешний ключ явно, то:

```
INSERT INTO products
  (name, description, price, catalog_id)
VALUES
  ('ASUS PRIME Z370-P', 'HDMI, SATA3, PCI Express 3.0,, USB 3.1', 9360.00, 2)//

SELECT id, name, price, catalog_id FROM products//
```

В таблицу попадет значение из запроса, триггер не будет вносить коррективы в параметры запроса. Итак, мы добились того, чтобы значение внешнего ключа корректировалось при вставке. Однако мы по-прежнему можем сделать поле **catalog_id** при помощи команды **UPDATE**:

```
UPDATE products SET catalog_id = NULL WHERE name = 'AMD RYZEN 5 1600'//
SELECT id, name, price, catalog_id FROM products//
```

Мы можем создать триггер и для команды **UPDATE**. Давайте при попытке назначить полю **catalog_id** значение будем оставлять текущее, если оно отлично от **NULL**, или заменять его не NULL-значением. Если и текущее и новое значения принимают значение **NULL**, будем назначать наименьшее значение из таблицы **catalogs**.

```
CREATE TRIGGER check_catalog_id_update BEFORE UPDATE ON products
FOR EACH ROW
BEGIN
  DECLARE cat_id INT;
  SELECT id INTO cat_id FROM catalogs ORDER BY id LIMIT 1;
  SET NEW.catalog_id = COALESCE(NEW.catalog_id, OLD.catalog_id, cat_id);
END//
```

```
UPDATE products SET catalog_id = NULL WHERE name = 'AMD RYZEN 5 1600'//
SELECT id, name, price, catalog_id FROM products//

UPDATE products SET catalog_id = 3 WHERE name = 'MSI B250M GAMING PRO'//
SELECT id, name, price, catalog_id FROM products//

UPDATE products SET catalog_id = NULL WHERE name = 'MSI B250M GAMING PRO'//
SELECT id, name, price, catalog_id FROM products//
```

Триггеры можно использовать, чтобы присваивать другим столбцам вычисляемые значения. Пусть у нас есть таблица **price**, которая содержит четыре столбца.

```
CREATE TABLE prices (
  id SERIAL PRIMARY KEY,
  processor DECIMAL (11,2) COMMENT 'Цена процессора',
  mother DECIMAL (11,2) COMMENT 'Цена мат.платы',
  memory DECIMAL (11,2) COMMENT 'Цена оперативной памяти',
  total DECIMAL (11,2) COMMENT 'Результирующая цена'
)//
```

Последний столбец **total** должен содержать сумму трех других, в этом случае мы можем заполнять его автоматически при помощи триггера.

```
CREATE TRIGGER auto_update_price_on_insert BEFORE INSERT ON prices
FOR EACH ROW
BEGIN
  SET NEW.total = NEW.processor + NEW.mother + NEW.memory;
END//
```

Давайте сразу создадим триггер и для обновления записи

```
CREATE TRIGGER auto_update_price_on_update BEFORE UPDATE ON prices
FOR EACH ROW
BEGIN
  SET NEW.total = NEW.processor + NEW.mother + NEW.memory;
END//
```

Тело запроса в нем будет точно такое же. Давайте попробуем вставить в таблицу **price** какие-либо записи.

```
INSERT INTO prices
  (processor, mother, memory)
VALUES
  (7890.00, 5060.00, 4800.00)//

INSERT INTO prices
  (processor, mother, memory)
VALUES
  (12700.00, 19310.00, 6800.00)//
```

Посмотрим содержимое таблицы prices:

```
SELECT * FROM prices//
```

Как видим, цена в столбце **total** обновляется автоматически. Впрочем, решить задачу можно с использованием STORED-столбцов. Давайте удалим таблицу **prices**:

```
DROP TABLE IF EXISTS prices//
```

И создадим ее снова с использованием STORED-столбца:

```
CREATE TABLE prices (
  id SERIAL PRIMARY KEY,
  processor DECIMAL (11,2) COMMENT 'Цена процессора',
  mother DECIMAL (11,2) COMMENT 'Цена мат.платы',
  memory DECIMAL (11,2) COMMENT 'Цена оперативной памяти',
    total DECIMAL (11,2) AS (processor + mother + memory) STORED COMMENT
'Результирующая цена'
)//
```

Повторно вставим записи:

```
INSERT INTO prices
  (processor, mother, memory)
VALUES
  (7890.00, 5060.00, 4800.00)//

INSERT INTO prices
  (processor, mother, memory)
VALUES
  (12700.00, 19310.00, 6800.00)//
```

Запросим содержимое таблицы **prices**:

```
SELECT * FROM prices//
```

Как видим, добиться точно такого же эффекта можно и без триггеров. Триггеры можно использовать не только для обновления и контроля состояния полей, но и для предотвращения операций.

Например, давайте добьемся, чтобы в таблице **catalogs** всегда присутствовала хотя бы одна запись. Мы просто не будем позволять удалять последнюю запись из таблицы. Для решения этой задачи удобно воспользоваться триггером **BEFORE DELETE**.

```
CREATE TRIGGER check_last_catalogs BEFORE DELETE ON catalogs
FOR EACH ROW BEGIN
  DECLARE total INT;
  SELECT COUNT(*) INTO total FROM catalogs;
  IF total <= 1 THEN
```

```
      SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'DELETE canceled';
  END IF;
END//
```

Триггер подсчитывает количество строк в таблице, и если оно меньше или равно единицы, запрещает дальнейшее выполнение запроса. Для этого мы генерируем свою собственную ошибку при помощи команды **SIGNAL SQLSTATE**.

Мы задаем код 45000, который предназначен для пользовательских ошибок. Их невозможно перехватить при помощи обработчиков. Давайте попробуем последовательно удалять записи из таблицы catalogs.

```
DELETE FROM catalogs LIMIT 1//
DELETE FROM catalogs LIMIT 1//
DELETE FROM catalogs LIMIT 1//
DELETE FROM catalogs LIMIT 1//
ERROR 1644 (45000): DELETE cancelled
```

Срабатывает триггер, выбрасывая ошибку с кодом 45000 и сообщением **DELETE cancelled**, которое мы задали внутри триггера. Убедимся, что таблица **catalogs** содержит по меньшей мере одну запись.

```
SELECT * FROM catalogs//
```

Так и есть, в таблице остается одна запись, и удалить ее не удастся, пока у нас есть триггер на операцию удаления.

Используемые источники

1. https://dev.mysql.com/doc/refman/5.7/en/create-procedure.html
2. https://dev.mysql.com/doc/refman/5.7/en/condition-handling.html
3. https://dev.mysql.com/doc/refman/5.7/en/cursors.html
4. Линн Бейли. Head First. Изучаем SQL. — СПб.: Питер, 2012. — 592 с.
5. Гровф, Джеймс Р., Вайнберг, Пол Н., Оппель, Эндрю Дж. SQL: полное руководство, 3-е изд. : Пер. с англ. — М.: ООО "И.Д. Вильямс", 2015. — 960 с.
6. Дейт К. Дж. SQL и реляционная теория. Как грамотно писать код на SQL. — Пер. с англ. — СПб.: Символ-Плюс, 2010. — 480 с.
7. Кузнецов М.В., Симдянов И.В. MySQL на примерах. — СПб.: БХВ-Петербург, 2007. — 592с.
8. Кузнецов М.В., Симдянов И.В. MySQL 5. — СПб.: БХВ-Петербург, 2006. — 1024с.
9. Дейт К. Дж. Введение в системы баз данных, 8-е издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2005. — 1328 с.
10. Карвин Б. Программирование баз данных SQL. Типичные ошибки и их устранение. — Рид Групп, 2011. — 336 с.