MySQL Error Handling in Stored Procedures

Summary: in this tutorial, you will learn how to use MySQL handler to handle errors encountered in stored procedures.

When an error occurs inside a stored procedure, it is important to handle it appropriately, such as continuing or exiting the current code block's execution, and issuing a meaningful error message.

MySQL provides an easy way to define handlers that handle from general conditions such as warnings or exceptions to specific conditions e.g., specific error codes.

Declaring a handler

To declare a handler, you use the `DECLARE HANDLER` statement as follows:

```
DECLARE action HANDLER FOR condition_value statement;
```

If a condition whose value matches the `condition_value` , MySQL will execute the `statement` and continue or exit the current code block based on the `action` .

The `action` accepts one of the following values:

- `CONTINUE` : the execution of the enclosing code block (`BEGIN` ... `END`) continues.
- `EXIT` : the execution of the enclosing code block, where the handler is declared, terminates.

The `condition_value` specifies a particular condition or a class of conditions that activate the handler. The `condition_value` accepts one of the following values:

- A MySQL error code.
- A standard `SQLSTATE` value. Or it can be an `SQLWARNING` , `NOTFOUND` or `SQLEXCEPTION` condition, which is shorthand for the class of `SQLSTATE` values. The `NOTFOUND` condition is used for a cursor or `SELECT INTO variable_list` statement.
- A named condition associated with either a MySQL error code or `SQLSTATE` value.

The `statement` could be a simple statement or a compound statement enclosing by the `BEGIN` and `END` keywords.

MySQL error handling examples

Let's take some examples of declaring handlers.

The following handler set the value of the `hasError` variable to 1 and continue the execution if an `SQLEXCEPTION` occurs

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
SET hasError = 1;
```

The following handler rolls back the previous operations, issues an error message, and exit the current code block in case an error occurs. If you declare it inside the `BEGIN END` block of a stored procedure, it will terminate the stored procedure immediately.

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
    ROLLBACK;
```

```
    SELECT 'An error has occurred, operation rollbacked and the stored procedure was terminated';
END;
```

The following handler sets the value of the `RowNotFound` variable to 1 and continues execution if there is no more row to fetch in case of a cursor or `SELECT INTO` statement:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND
SET RowNotFound = 1;
```

If a duplicate key error occurs, the following handler issues an error message and continues execution.

```
DECLARE CONTINUE HANDLER FOR 1062
SELECT 'Error, duplicate key occurred';
```

MySQL handler example in stored procedures

First, create a new table named `SupplierProducts` for the demonstration:

```
CREATE TABLE SupplierProducts (
    supplierId INT,
    productId INT,
    PRIMARY KEY (supplierId , productId)
);
```

The table `SupplierProducts` stores the relationships between the table suppliers and products. Each supplier may provide many products and each product can be provided by many suppliers. For the sake of simplicity, we don't create `Products` and `Suppliers` tables, as well as the foreign keys in the `SupplierProducts` table.

Second, create a stored procedure that inserts product id and supplier id into the `SupplierProducts` table:

```
CREATE PROCEDURE InsertSupplierProduct(
    IN inSupplierId INT,
    IN inProductId INT
)
BEGIN
    -- exit if the duplicate key occurs
    DECLARE EXIT HANDLER FOR 1062
    BEGIN
        SELECT CONCAT('Duplicate key (',inSupplierId,',',inProductId,') occurred') AS message;
    END;

    -- insert a new row into the SupplierProducts
    INSERT INTO SupplierProducts(supplierId,productId)
    VALUES(inSupplierId,inProductId);

    -- return the products supplied by the supplier id
    SELECT COUNT(*)
    FROM SupplierProducts
    WHERE supplierId = inSupplierId;

END$$
```

```
DELIMITER ;
```

How it works.

The following exit handler terminates the stored procedure whenever a duplicate key occurs (with code 1062). In addition, it returns an error message.

```
DECLARE EXIT HANDLER FOR 1062
BEGIN
    SELECT CONCAT('Duplicate key (',supplierId,',',productId,') occurred') AS message;
END;
```

This statement inserts a row into the `SupplierProducts` table. If a duplicate key occurs, the code in the handler section will execute.

```
INSERT INTO SupplierProducts(supplierId,productId)
VALUES(supplierId,productId);
```

Third, call the `InsertSupplierProduct()` to insert some rows into the `SupplierProducts` table:

```
CALL InsertSupplierProduct(1,1);
CALL InsertSupplierProduct(1,2);
CALL InsertSupplierProduct(1,3);
```

Fourth, attempt to insert a row whose values already exist in the `SupplierProducts` table:

```
CALL InsertSupplierProduct(1,3);
```

Here is the error message:

```
+-----------------------------+
| message                     |
+-----------------------------+
| Duplicate key (1,3) occurred |
+-----------------------------+
1 row in set (0.01 sec)
```

Because the handler is an `EXIT` handler, the last statement does not execute:

```
SELECT COUNT(*)
FROM SupplierProducts
WHERE supplierId = inSupplierId;
```

If you change the `EXIT` in the handler declaration to `CONTINUE`, you will also get the number of products provided by the supplier:

```
DROP PROCEDURE IF EXISTS InsertSupplierProduct;

DELIMITER $$

CREATE PROCEDURE InsertSupplierProduct(
    IN inSupplierId INT,
```

```
    IN inProductId INT
)
BEGIN
    -- exit if the duplicate key occurs
    DECLARE CONTINUE HANDLER FOR 1062
    BEGIN
        SELECT CONCAT('Duplicate key (',inSupplierId,',',inProductId,') occurred') AS message;
    END;

    -- insert a new row into the SupplierProducts
    INSERT INTO SupplierProducts(supplierId,productId)
    VALUES(inSupplierId,inProductId);

    -- return the products supplied by the supplier id
    SELECT COUNT(*)
    FROM SupplierProducts
    WHERE supplierId = inSupplierId;

END$$

DELIMITER ;
```

Finally, call the stored procedure again to see the effect of the `CONTINUE` handler:

```
CALL InsertSupplierProduct(1,3);
```

Here is the output:

```
+----------+
| COUNT(*) |
+----------+
|        3 |
+----------+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.02 sec)
```

MySQL handler precedence

In case you have multiple handlers that handle the same error, MySQL will call the most specific handler to handle the error first based on the following rules:

- An error always maps to a MySQL error code because in MySQL it is the most specific.
- An `SQLSTATE` may map to many MySQL error codes, therefore, it is less specific.
- An `SQLEXCPETION` or an `SQLWARNING` is the shorthand for a class of `SQLSTATES` values so it is the most generic.

Based on the handler precedence rules, MySQL error code handler, `SQLSTATE` handler and `SQLEXCEPTION` takes the first, second and third precedence.

Suppose that we have three handlers in the handlers in the stored procedure `insert_article_tags_3` :

```
DROP PROCEDURE IF EXISTS InsertSupplierProduct;

DELIMITER $$

CREATE PROCEDURE InsertSupplierProduct(
    IN inSupplierId INT,
    IN inProductId INT
)
BEGIN
    -- exit if the duplicate key occurs
    DECLARE EXIT HANDLER FOR 1062 SELECT 'Duplicate keys error encountered' Message;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION SELECT 'SQLException encountered' Message;
    DECLARE EXIT HANDLER FOR SQLSTATE '23000' SELECT 'SQLSTATE 23000' ErrorCode;

    -- insert a new row into the SupplierProducts
    INSERT INTO SupplierProducts(supplierId,productId)
    VALUES(inSupplierId,inProductId);

    -- return the products supplied by the supplier id
    SELECT COUNT(*)
    FROM SupplierProducts
    WHERE supplierId = inSupplierId;

END$$

DELIMITER ;
```

Call the stored procedure to insert a duplicate key:

```
CALL InsertSupplierProduct(1,3);
```

Here is the output:

```
+---------------------------------+
| Message                         |
+---------------------------------+
| Duplicate keys error encountered |
+---------------------------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.01 sec)
```

As you see the MySQL error code handler is called.

Using a named error condition

Let's start with an error handler declaration.

```
DELIMITER $$

CREATE PROCEDURE TestProc()
BEGIN
```

```
    DECLARE EXIT HANDLER FOR 1146
    SELECT 'Please create table abc first' Message;

    SELECT * FROM abc;
END$$

DELIMITER ;
```

What does the number `1146` really mean? Imagine you have stored procedures polluted with these numbers all over places; it will be difficult to understand and maintain the code.

Fortunately, MySQL provides you with the `DECLARE CONDITION` statement that declares a named error condition, which associates with a condition.

Here is the syntax of the `DECLARE CONDITION` statement:

```
DECLARE condition_name CONDITION FOR condition_value;
```

The `condition_value` can be a MySQL error code such as `1146` or a `SQLSTATE` value. The `condition_value` is represented by the `condition_name` .

After the declaration, you can refer to `condition_name` instead of `condition_value` .

So you can rewrite the code above as follows:

```
DROP PROCEDURE IF EXISTS TestProc;
```

```
DELIMITER $$

CREATE PROCEDURE TestProc()
BEGIN
    DECLARE TableNotFound CONDITION for 1146 ;

    DECLARE EXIT HANDLER FOR TableNotFound
        SELECT 'Please create table abc first' Message;
    SELECT * FROM abc;
END$$

DELIMITER ;
```

As you can see, the code is more obviously and readable than the previous one. Notice that the condition declaration must appear before handler or cursor declarations.

In this tutorial, you have learned how to use MySQL handlers to handle exception or errors occurred in stored procedures.