Create User-defined Functions (Database Engine)

This topic describes how to create a Transact-SQL user-defined function in SQL Server 2012 by using Transact-SQL.

In This Topic

Before You Begin

Limitations and Restrictions

- User-defined functions cannot be used to perform actions that modify the database state.
- User-defined functions cannot contain an OUTPUT INTO clause that has a table as its target.
- The following Service Broker statements cannot be included in the definition of a Transact-SQL user-defined function:
  - BEGIN DIALOG CONVERSATION
  - END CONVERSATION
  - GET CONVERSATION GROUP
  - MOVE CONVERSATION
  - RECEIVE
  - SEND
- User-defined functions can be nested; that is, one user-defined function can call another. The nesting level is incremented when the called function starts execution, and decremented when the called function finishes execution. User-defined functions can be nested up to 32 levels. Exceeding the maximum levels of nesting causes the whole calling function chain to fail. Any reference to managed code from a Transact-SQL user-defined function counts as one level against the 32-level

nesting limit. Methods invoked from within managed code do not count against this limit.

Security

Permissions

Requires CREATE FUNCTION permission in the database and ALTER permission on the schema in which the function is being created. If the function specifies a user-defined type, requires EXECUTE permission on the type.

Scalar Functions

The following example creates a multistatement scalar function. The function takes one input value, a ProductID, and returns a single data value, the aggregated quantity of the specified product in inventory.

Transact-SQL
```
USE AdventureWorks2012;
GO
IF OBJECT_ID (N'dbo.ufnGetInventoryStock', N'FN') IS NOT NULL
    DROP FUNCTION ufnGetInventoryStock;
GO
CREATE FUNCTION dbo.ufnGetInventoryStock(@ProductID int)
RETURNS int
AS
-- Returns the stock level for the product.
BEGIN
    DECLARE @ret int;
    SELECT @ret = SUM(p.Quantity)
    FROM Production.ProductInventory p
    WHERE p.ProductID = @ProductID
        AND p.LocationID = '6';
     IF (@ret IS NULL)
        SET @ret = 0;
    RETURN @ret;
END;
GO
```

The following example uses the ufnGetInventoryStock function to return the current inventory quantity for products that have a ProductModelID between 75 and 80.

Transact-SQL
```
USE AdventureWorks2012;
GO
SELECT ProductModelID, Name, dbo.ufnGetInventoryStock(ProductID)AS
CurrentSupply
FROM Production.Product
```

```
WHERE ProductModelID BETWEEN 75 and 80;
GO
```

Table-Valued Functions

The following example creates an inline table-valued function. The function takes one input parameter, a customer (store) ID, and returns the columns ProductID, Name, and the aggregate of year-to-date sales as YTD Total for each product sold to the store.

Transact-SQL
```
USE AdventureWorks2012;
GO
IF OBJECT_ID (N'Sales.ufn_SalesByStore', N'IF') IS NOT NULL
    DROP FUNCTION Sales.ufn_SalesByStore;
GO
CREATE FUNCTION Sales.ufn_SalesByStore (@storeid int)
RETURNS TABLE
AS
RETURN
(
    SELECT P.ProductID, P.Name, SUM(SD.LineTotal) AS 'Total'
    FROM Production.Product AS P
    JOIN Sales.SalesOrderDetail AS SD ON SD.ProductID = P.ProductID
    JOIN Sales.SalesOrderHeader AS SH ON SH.SalesOrderID =
SD.SalesOrderID
    JOIN Sales.Customer AS C ON SH.CustomerID = C.CustomerID
    WHERE C.StoreID = @storeid
    GROUP BY P.ProductID, P.Name
);
GO
```

The following example invokes the function and specifies customer ID 602.

Transact-SQL
```
SELECT * FROM Sales.ufn_SalesByStore (602);
```

The following example creates a table-valued function. The function takes a single input parameter, an EmployeeID and returns a list of all the employees who report to the specified employee directly or indirectly. The function is then invoked specifying employee ID 109.

Transact-SQL
```
USE AdventureWorks2012;
GO
IF OBJECT_ID (N'dbo.ufn_FindReports', N'TF') IS NOT NULL
    DROP FUNCTION dbo.ufn_FindReports;
GO
CREATE FUNCTION dbo.ufn_FindReports (@InEmpID INTEGER)
```

```
RETURNS @retFindReports TABLE
(
    EmployeeID int primary key NOT NULL,
    FirstName nvarchar(255) NOT NULL,
    LastName nvarchar(255) NOT NULL,
    JobTitle nvarchar(50) NOT NULL,
    RecursionLevel int NOT NULL
)
--Returns a result set that lists all the employees who report to the
--specific employee directly or indirectly.*/
AS
BEGIN
WITH EMP_cte(EmployeeID, OrganizationNode, FirstName, LastName, JobTitle,
RecursionLevel) -- CTE name and columns
    AS (
        SELECT e.BusinessEntityID, e.OrganizationNode, p.FirstName,
p.LastName, e.JobTitle, 0 -- Get the initial list of Employees for
Manager n
        FROM HumanResources.Employee e
                       INNER JOIN Person.Person p
                       ON p.BusinessEntityID = e.BusinessEntityID
        WHERE e.BusinessEntityID = @InEmpID
        UNION ALL
        SELECT e.BusinessEntityID, e.OrganizationNode, p.FirstName,
p.LastName, e.JobTitle, RecursionLevel + 1 -- Join recursive member to
anchor
        FROM HumanResources.Employee e
            INNER JOIN EMP_cte
            ON e.OrganizationNode.GetAncestor(1) =
EMP_cte.OrganizationNode
                       INNER JOIN Person.Person p
                       ON p.BusinessEntityID = e.BusinessEntityID
        )
-- copy the required columns to the result of the function
   INSERT @retFindReports
   SELECT EmployeeID, FirstName, LastName, JobTitle, RecursionLevel
   FROM EMP_cte
   RETURN
END;
GO
-- Example invocation
SELECT EmployeeID, FirstName, LastName, JobTitle, RecursionLevel
FROM dbo.ufn_FindReports(1);

GO
```

User-Defined Functions

Like functions in programming languages, SQL Server user-defined functions are routines that accept parameters, perform an action, such as a complex calculation, and return the result of that action as a value. The return value can either be a single scalar value or a result set.

In This Topic

User-defined Function Benefits

The benefits of using user-defined functions in SQL Server are:

- They allow modular programming.

  You can create the function once, store it in the database, and call it any number of times in your program. User-defined functions can be modified independently of the program source code.

- They allow faster execution.

  Similar to stored procedures, Transact-SQL user-defined functions reduce the compilation cost of Transact-SQL code by caching the plans and reusing them for repeated executions. This means the user-defined function does not need to be reparsed and reoptimized with each use resulting in much faster execution times.

  CLR functions offer significant performance advantage over Transact-SQL functions for computational tasks, string manipulation, and business logic. Transact-SQL functions are better suited for data-access intensive logic.

- They can reduce network traffic.

  An operation that filters data based on some complex constraint that cannot be expressed in a single scalar expression can be expressed as a function. The function can then invoked in the WHERE clause to reduce the number or rows sent to the client.

Note

Transact-SQL user-defined functions in queries can only be executed on a single thread (serial execution plan).

Types of Functions

Scalar Function

User-defined scalar functions return a single data value of the type defined in the RETURNS clause. For an inline scalar function, there is no function body; the scalar value is the result of a single statement. For a multistatement scalar function, the function body, defined in a BEGIN...END block, contains a series of Transact-SQL statements that return the single value. The return type can be any data type except text, ntext, image, cursor, and timestamp.

Table-Valued Functions

User-defined table-valued functions return a table data type. For an inline table-valued function, there is no function body; the table is the result set of a single SELECT statement.

System Functions

SQL Server provides many system functions that you can use to perform a variety of operations. They cannot be modified. For more information, see Built-in Functions (Transact-SQL), System Stored Functions (Transact-SQL), and Dynamic Management Views and Functions (Transact-SQL).

Guidelines

Transact-SQL errors that cause a statement to be canceled and continue with the next statement in the module (such as triggers or stored procedures) are treated differently inside a function. In functions, such errors cause the execution of the function to stop. This in turn causes the statement that invoked the function to be canceled.

The statements in a BEGIN...END block cannot have any side effects. Function side effects are any permanent changes to the state of a resource that has a scope outside the function such as a modification to a database table. The only changes that can be made by the statements in the function are changes to objects local to the function, such as local cursors or variables. Modifications to database tables, operations on cursors that are not local to the function, sending e-mail, attempting a catalog modification, and generating a result set that is returned to the user are examples of actions that cannot be performed in a function.

Note

If a CREATE FUNCTION statement produces side effects against resources that do not exist when the CREATE FUNCTION statement is issued, SQL Server executes the statement. However, SQL Server does not execute the function when it is invoked.

The number of times that a function specified in a query is actually executed can vary between execution plans built by the optimizer. An example is a function invoked by a subquery in a WHERE clause. The number of times the subquery and its function is executed can vary with different access paths chosen by the optimizer.

Valid Statements in a Function

The types of statements that are valid in a function include:

- DECLARE statements can be used to define data variables and cursors that are local to the function.
- Assignments of values to objects local to the function, such as using SET to assign values to scalar and table local variables.
- Cursor operations that reference local cursors that are declared, opened, closed, and deallocated in the function. FETCH statements that return data to the client are not allowed. Only FETCH statements that assign values to local variables using the INTO clause are allowed.
- Control-of-flow statements except TRY...CATCH statements.
- SELECT statements containing select lists with expressions that assign values to variables that are local to the function.
- UPDATE, INSERT, and DELETE statements modifying table variables that are local to the function.
- EXECUTE statements calling an extended stored procedure.

Built-in System Functions

The following nondeterministic built-in functions can be used in Transact-SQL user-defined functions.

CURRENT_TIMESTAMP	@@MAX_CONNECTIONS
GET_TRANSMISSION_STATUS	@@PACK_RECEIVED
GETDATE	@@PACK_SENT
GETUTCDATE	@@PACKET_ERRORS
@@CONNECTIONS	@@TIMETICKS
@@CPU_BUSY	@@TOTAL_ERRORS
@@DBTS	@@TOTAL_READ

@@IDLE	@@TOTAL_WRITE
@@IO_BUSY	

The following nondeterministic built-in functions cannot be used in Transact-SQL user-defined functions.

NEWID	RAND
NEWSEQUENTIALID	TEXTPTR

For a list of deterministic and nondeterministic built-in system functions, see Deterministic and Nondeterministic Functions.

Schema-Bound Functions

CREATE FUNCTION supports a SCHEMABINDING clause that binds the function to the schema of any objects it references, such as tables, views, and other user-defined functions. An attempt to alter or drop any object referenced by a schema-bound function fails.

These conditions must be met before you can specify SCHEMABINDING in CREATE FUNCTION:

- All views and user-defined functions referenced by the function must be schema-bound.
- All objects referenced by the function must be in the same database as the function. The objects must be referenced using either one-part or two-part names.
- You must have REFERENCES permission on all objects (tables, views, and user-defined functions) referenced in the function.

You can use ALTER FUNCTION to remove the schema binding. The ALTER FUNCTION statement should redefine the function without specifying WITH SCHEMABINDING.

Specifying Parameters

A user-defined function takes zero or more input parameters and returns either a scalar value or a table. A function can have a maximum of 1024 input parameters. When a parameter of the function has a default value, the keyword DEFAULT must be specified when calling the function to get the default value. This behavior is different from parameters with default values in user-defined stored procedures in which omitting the parameter also implies the default value. User-defined functions do not support output parameters.