T-SQL
SQL-SERVER

Tabla de Contenidos

FUNCIONES

CREATE FUNCTION

Crea una función definida por el usuario, que es una rutina guardada de Transact-SQL que devuelve un valor. Las funciones definidas por el usuario no se pueden utilizar para realizar un conjunto de acciones que modifican el estado global de una base de datos. Las funciones definidas por el usuario, como las funciones de sistema, se pueden llamar desde una consulta. También se pueden ejecutar mediante una instrucción EXECUTE como procedimientos almacenados.

Las funciones definidas por el usuario se modifican utilizando ALTER FUNCTION y se eliminan mediante DROP FUNCTION.

Sintaxis

Funciones escalares

CREATE FUNCTION [*owner_name***.**] *function_name*
([{ *@parameter_name* [AS] *scalar_parameter_data_type* [= *default*] } [**,***...n*]] **)**

RETURNS *scalar_return_data_type*

[WITH < function_option> [[**,**] ...*n*]]

[AS]

BEGIN
function_body
RETURN *scalar_expression*
END

Funciones de valores de tabla en línea

CREATE FUNCTION [*owner_name***.**] *function_name*
([{ *@parameter_name* [AS] *scalar_parameter_data_type* [= *default*] } [**,***...n*]] **)**

RETURNS TABLE

[WITH < function_option > [[**,**] ...*n*]]

[AS]

RETURN [**(**] *select-stmt* [**)**]

Funciones de valores de tabla de múltiples instrucciones

CREATE FUNCTION [*owner_name***.**] *function_name*
([{ *@parameter_name* [AS] *scalar_parameter_data_type* [= *default*] } [**,***...n*]] **)**

RETURNS *@return_variable* TABLE < table_type_definition >

[WITH < function_option > [[**,**] ...*n*]]

[AS]

BEGIN
function_body
RETURN
END

< function_option > ::=
{ ENCRYPTION | SCHEMABINDING }

< table_type_definition > :: =
({ *column_definition* | *table_constraint* } [**,***...n*] **)**

Argumentos

owner_name

Nombre del Id. de usuario que posee la función definida por el usuario. *owner_name* debe ser un Id. de usuario existente.

function_name

Nombre de la función definida por el usuario. Los nombres de funciones deben seguir las reglas de los identificadores y deben ser únicos en la base de datos y para su propietario.

@parameter_name

Es un parámetro de la función definida por el usuario. En una instrucción CREATE FUNCTION se pueden declarar uno o varios parámetros. Una función puede tener un máximo de 1.024 parámetros. El usuario debe proporcionar el valor de cada parámetro declarado cuando se ejecuta la función, a menos que se defina un valor predeterminado para el parámetro. Cuando un parámetro de la función toma un valor predeterminado, debe especificarse la palabra clave "default" al llamar a la función para poder obtener el valor predeterminado. Este comportamiento es diferente del de los parámetros con valores

predeterminados de los procedimientos almacenados, para los cuales omitir el parámetro implica especificar el valor predeterminado.

Especifique un nombre de parámetro con un signo (@) como el primer carácter. Los nombres de los parámetros deben cumplir las reglas de los identificadores. Los parámetros son locales para la función; los mismos nombres de parámetro se pueden utilizar en otras funciones. Los parámetros sólo pueden ocupar el lugar de constantes; no se pueden utilizar en lugar de nombres de tablas, nombres de columnas o nombres de otros objetos de base de datos.

scalar_parameter_data_type

Es el tipo de datos del parámetro. Todos los tipos de datos escalares, incluidos **bigint** y **sql_variant**, se pueden utilizar como un parámetro para las funciones definidas por el usuario. No se admiten el tipo de datos **timestamp** ni los tipos de datos definidos por el usuario. No se pueden especificar tipos no escalares, como cursor y tabla.

scalar_return_data_type

Es el valor de retorno de una función escalar definida por el usuario. *scalar_return_data_type* puede ser de cualquiera de los tipos de datos escalares compatibles con SQL Server, excepto **text**, **ntext**, **image** y **timestamp**.

scalar_expression

Especifica el valor escalar que devuelve la función escalar.

TABLE

Especifica que el valor de retorno de la función que devuelve valores de tabla es una tabla.

En las funciones que devuelven valores de tabla en línea, el valor de retorno de TABLE se define mediante una única instrucción SELECT. Las funciones en línea no tienen asociadas variables de retorno.

En las funciones que devuelven valores de tabla de múltiples instrucciones, *@return_variable* es una variable TABLE, que se utiliza para almacenar y acumular las filas que se deben devolver como valor de la función.

function_body

Especifica que una serie de instrucciones Transact-SQL, que juntas no producen ningún efecto secundario, definen el valor de la función. *function_body* sólo se utiliza en funciones escalares y funciones que devuelven valores de tabla de múltiples instrucciones.

En las funciones escalares, *function_body* es una serie de instrucciones Transact-SQL que juntas dan como resultado un valor escalar.

En las funciones que devuelven valores de tabla de múltiples instrucciones, *function_body* es una serie de instrucciones Transact-SQL que rellena una variable de retorno de tabla.

select-stmt

Es la instrucción SELECT individual que define el valor de retorno de una función en línea que devuelve valores de tabla.

ENCRYPTION

Indica que SQL Server cifra las columnas de la tabla del sistema que contienen el texto de la instrucción CREATE FUNCTION. El uso de ENCRYPTION impide que la función se publique como parte de la duplicación de SQL Server.

SCHEMABINDING

Especifica que la función está enlazada a los objetos de base de datos a los que hace referencia. Si se crea una función con la opción SCHEMABINDING, los objetos de base de datos a los que hace referencia la función no se pueden modificar (mediante la instrucción ALTER) ni eliminar (mediante la instrucción DROP).

El enlace de la función a los objetos a los que hace referencia sólo se elimina cuando se ejecuta una de estas dos acciones:

- Se quita la función.
- La función se cambia (mediante la instrucción ALTER) sin especificar la opción SCHEMABINDING.

Una función se puede enlazar a esquema sólo si se cumplen las siguientes condiciones:

- Las funciones definidas por el usuario y las vistas a las que hace referencia la función también están enlazadas al esquema.
- La función no hace referencia a los objetos utilizando un nombre en dos partes.
- La función y los objetos a los que hace referencia pertenecen a la misma base de datos.
- El usuario que ha ejecutado la instrucción CREATE FUNCTION tiene permisos REFERENCES sobre todos los objetos de base de datos a los que hace referencia la función.

Si no se cumplen las condiciones anteriores, se producirá un error al ejecutar la instrucción CREATE FUNCTION especificando la opción SCHEMABINDING.

Observaciones

Las funciones definidas por el usuario son de valores de tabla o de valores escalares. Son funciones de valores escalares si la cláusula RETURNS especificó uno de los tipos de datos escalares. Las funciones de valores escalares se pueden definir utilizando varias instrucciones Transact-SQL.

Son funciones de valores de tabla si la cláusula RETURNS especificó TABLE. Según cómo se haya definido el cuerpo de la función, las funciones de valores de tabla se pueden clasificar como en funciones en línea o de múltiples instrucciones.

Si la cláusula RETURNS especifica TABLE sin una lista de columnas, la función es en línea. Las funciones en línea son funciones de valores de tabla definidas con una única instrucción SELECT como parte del cuerpo de la función. Las columnas, incluidos los tipos de datos, de la tabla que devuelve la función, proceden de la lista SELECT de la instrucción SELECT que define la función.

Si la cláusula RETURNS especifica un tipo TABLE con columnas y sus tipos de datos, se trata de una función de valores de tabla de múltiples instrucciones.

El cuerpo de una función de múltiples instrucciones permite las siguientes instrucciones. Las instrucciones no incluidas en la lista no se permiten en el cuerpo de una función:

- Instrucciones de asignación.
- Las instrucciones de control de flujo.
- Instrucciones DECLARE que definen variables de datos y cursores que son locales a la función.
- Instrucciones SELECT que contienen listas de selección con expresiones que asignan valores a las variables locales para la función.
- Operaciones de cursor que hacen referencia a cursores locales que se declaran, abren, cierran y cuya asignación se cancela en la función. Sólo se permiten las instrucciones FETCH que asignan valores a las variables locales mediante la cláusula INTO; no se permiten las instrucciones FETCH que devuelven los datos al cliente.
- Instrucciones INSERT, UPDATE y DELETE que modifican las variables **table** locales para la función.
- Instrucciones EXECUTE que llaman a procedimientos almacenados extendidos.

Determinismo de funciones y efectos secundarios

Las funciones son deterministas o no deterministas. Son deterministas cuando devuelven siempre el mismo resultado cada vez que se les llama con un conjunto específico de valores de entrada. Son no deterministas cuando es posible que devuelvan distintos resultados cada vez que se les llama con un mismo conjunto específico de valores de entrada.

Las funciones no deterministas pueden provocar efectos secundarios. Los efectos secundarios son cambios en el estado general de la base de datos, como una actualización en una tabla de la base de datos o en algún recurso externo, como un archivo o la red (por ejemplo, modificar un archivo o enviar un mensaje de correo electrónico).

No se permiten las funciones no deterministas integradas en el cuerpo de la funciones definidas por el usuario. Son las siguientes:

@@CONNECTIONS	@@TOTAL_ERRORS
@@CPU_BUSY	@@TOTAL_READ
@@IDLE	@@TOTAL_WRITE
@@IO_BUSY	GETDATE
@@MAX_CONNECTIONS	GETUTCDATE
@@PACK_RECEIVED	NEWID
@@PACK_SENT	RAND
@@PACKET_ERRORS	TEXTPTR
@@TIMETICKS	

Aunque no se permiten las funciones no deterministas en el cuerpo de las funciones definidas por el usuario, estas últimas pueden provocar efectos secundarios si se llaman desde procedimientos almacenados extendidos.

Las funciones que llaman a procedimientos almacenados extendidos se consideran no deterministas, debido a que estos procedimientos pueden provocar efectos secundarios en la base de datos. Cuando las funciones definidas por el usuario llaman a procedimientos almacenados extendidos que pueden provocar efectos secundarios en la base de datos, no es posible confiar en la ejecución de la función ni en que se produzca un conjunto de resultados consistente.

Llamar a procedimientos almacenados extendidos desde funciones

Cuando se llama a un procedimiento almacenado extendido desde una función, no se puede devolver al cliente el conjunto de resultados. Cualquier API ODS que devuelva conjuntos de resultados al cliente devolverá FAIL. El procedimiento almacenado extendido no puede

volver a conectar con Microsoft® SQL Server™; sin embargo, no debería intentar combinar la misma transacción que la función que invocó al procedimiento almacenado extendido.

De la misma manera que en las invocaciones desde un proceso por lotes o un procedimiento almacenado, el procedimiento almacenado extendido se ejecutará en el contexto de la cuenta de seguridad de Windows® bajo la que se ejecuta SQL Server. El propietario del procedimiento almacenado lo debería tener en cuenta al otorgar permisos EXECUTE a los usuarios.

Invocación a funciones

Las funciones de valores escalares se pueden llamar en aquellos lugares donde se utilizan expresiones escalares, incluidas las columnas calculadas y las definiciones de restricciones CHECK. Al llamar a funciones de valores escalares, utilice como mínimo el nombre de dos partes de la función.

[database_name.]owner_name.function_name ([argument_expr][,...])

Si utiliza una función definida por el usuario para definir una columna calculada, la calidad determinista de la función también define si se creará un índice en esa columna calculada. En una columna calculada, se puede crear un índice que utiliza una función sólo si la función es determinista. Una función es determinista si siempre devuelve el mismo valor con los mismos datos de entrada.

Las funciones de valores de tabla se pueden llamar utilizando un nombre en una sola parte.

[database_name.][owner_name.]function_name ([argument_expr][,...])

Las funciones de la tabla de sistema incluidas en Microsoft® SQL Server™ 2000 deben invocarse con el prefijo '::' antes del nombre de la función.

```
SELECT *
FROM ::fn_helpcollations()
```

Los errores de Transact-SQL que provocan la detención de una instrucción y la continuación con la instrucción siguiente en un procedimiento almacenado se tratan de manera distinta dentro de una función. En las funciones, esos errores provocarán la detención de la función. Eso a su vez provocará la detención de la instrucción a la que invocó la función.

De forma predeterminada, los permisos CREATE FUNCTION se conceden a los miembros de la función fija de servidor **sysadmin** y a las funciones fijas de base de datos **db_owner** y **db_ddladmin**. Los miembros de **sysadmin** y **db_owner** pueden conceder permisos CREATE FUNCTION a otros inicios de sesión mediante la instrucción GRANT.

Los propietarios de funciones tienen permiso EXECUTE sobre sus funciones. Los demás usuarios no tienen permisos EXECUTE a menos que se les conceda permisos EXECUTE sobre la función específica.

Para poder crear o modificar tablas con referencias a funciones definidas por el usuario en las cláusulas CONSTRAINT o DEFAULT, o en la definición de la columna calculada, el usuario también debe tener permisos REFERENCES sobre las funciones.

Ejemplos

A. Función de valores escalares definida por el usuario que calcula la semana ISO

En este ejemplo, una función definida por el usuario, ISOweek, toma un argumento de fecha y calcula el número de semana ISO. Para que esta función realice el cálculo correctamente, se debe llamar a SET DATEFIRST 1 antes de llamar a la función.

```
CREATE FUNCTION ISOweek  (@DATE datetime)
RETURNS int
AS
BEGIN
   DECLARE @ISOweek int
   SET @ISOweek= DATEPART(wk,@DATE)+1
      -DATEPART(wk,CAST(DATEPART(yy,@DATE) as CHAR(4))+'0104')
--Special cases: Jan 1-3 may belong to the previous year
   IF (@ISOweek=0)
      SET @ISOweek=dbo.ISOweek(CAST(DATEPART(yy,@DATE)-1
         AS CHAR(4))+'12'+ CAST(24+DATEPART(DAY,@DATE) AS CHAR(2)))+1
--Special case: Dec 29-31 may belong to the next year
   IF ((DATEPART(mm,@DATE)=12) AND
      ((DATEPART(dd,@DATE)-DATEPART(dw,@DATE))>= 28))
      SET @ISOweek=1
   RETURN(@ISOweek)
END
```

Ésta es la llamada a la función. Observe que el valor de DATEFIRST es 1.

```
SET DATEFIRST 1
SELECT master.dbo.ISOweek('12/26/1999') AS 'ISO Week'
```

Éste es el conjunto de resultados:

```
ISO Week
----------------
52
```

B. Función de valores de tabla en línea

Este ejemplo devuelve una función de valores de tabla en línea.

```
USE pubs
GO
```

```
CREATE FUNCTION SalesByStore (@storeid varchar(30))
RETURNS TABLE
AS
RETURN (SELECT title, qty
      FROM sales s, titles t
      WHERE s.stor_id = @storeid and
      t.title_id = s.title_id)
```

C. Función de valores de tabla de múltiples instrucciones

Tomemos una tabla que representa una relación jerárquica:

```
CREATE TABLE employees (empid nchar(5) PRIMARY KEY,
      empname nvarchar(50),
      mgrid nchar(5) REFERENCES employees(empid),
      title nvarchar(30)
      )
```

La función de valores de tabla fn_FindReports(InEmpID) que, dado un Id. de empleado, devuelve una tabla que corresponde a todos los empleados que informan a dicho empleado directa o indirectamente. Esta lógica no se puede expresar en una sola consulta y es un buen ejemplo para utilizarla como función definida por el usuario.

```
CREATE FUNCTION fn_FindReports (@InEmpId nchar(5))
RETURNS @retFindReports TABLE (empid nchar(5) primary key,
   empname nvarchar(50) NOT NULL,
   mgrid nchar(5),
   title nvarchar(30))
/*Returns a result set that lists all the employees who report to given
employee directly or indirectly.*/
AS
BEGIN
   DECLARE @RowsAdded int
   -- table variable to hold accumulated results
   DECLARE @reports TABLE (empid nchar(5) primary key,
      empname nvarchar(50) NOT NULL,
      mgrid nchar(5),
      title nvarchar(30),
      processed tinyint default 0)
-- initialize @Reports with direct reports of the given employee
   INSERT @reports
   SELECT empid, empname, mgrid, title, 0
   FROM employees
   WHERE empid = @InEmpId
   SET @RowsAdded = @@rowcount
   -- While new employees were added in the previous iteration
   WHILE @RowsAdded > 0
   BEGIN
      /*Mark all employee records whose direct reports are going to be
   found in this iteration with processed=1.*/
      UPDATE @reports
      SET processed = 1
      WHERE processed = 0
      -- Insert employees who report to employees marked 1.
```

```
      INSERT @reports
      SELECT e.empid, e.empname, e.mgrid, e.title, 0
      FROM employees e, @reports r
      WHERE e.mgrid=r.empid and e.mgrid <> e.empid and r.processed = 1
      SET @RowsAdded = @@rowcount
      /*Mark all employee records whose direct reports have been found
   in this iteration.*/
      UPDATE @reports
      SET processed = 2
      WHERE processed = 1
   END

   -- copy to the result of the function the required columns
   INSERT @retFindReports
   SELECT empid, empname, mgrid, title
   FROM @reports
   RETURN
END
GO

-- Example invocation
SELECT *
FROM fn_FindReports('11234')
GO
```

ALTER FUNCTION

Modifica una función existente definida por el usuario, creada anteriormente por la ejecución de la instrucción CREATE FUNCTION, sin cambiar los permisos y sin que afecte a ninguna otra función, procedimiento almacenado o desencadenador dependientes.

Sintaxis

Funciones escalares

ALTER FUNCTION [*owner_name***.**] *function_name*
   ([{ *@parameter_name scalar_parameter_data_type* [= *default*] } [**,***...n*]] **)**

RETURNS *scalar_return_data_type*

[WITH < function_option > [**,***...n*]]

[AS]

BEGIN
   function_body
   RETURN *scalar_expression*
END

Funciones de valores de tabla en línea

ALTER FUNCTION [*owner_name***.**] *function_name*
([{ *@parameter_name scalar_parameter_data_type* [= *default*] } [**,**...*n*]] **)**

RETURNS TABLE

[WITH < function_option > [**,**...*n*]]

[AS]

RETURN [**(**] *select-stmt* [**)**]

Funciones de valores de tabla de múltiples instrucciones

ALTER FUNCTION [*owner_name***.**] *function_name*
([{ *@parameter_name scalar_parameter_data_type* [= *default*] } [**,**...*n*]] **)**

RETURNS *@return_variable* TABLE < table_type_definition >

[WITH < function_option > [**,**...*n*]]

[AS]

BEGIN
function_body
RETURN
END

< function_option > ::=
{ ENCRYPTION | SCHEMABINDING }

< table_type_definition > :: =
({ *column_definition* | *table_constraint* } [**,**...*n*] **)**

Argumentos

owner_name

Es el nombre del Id. de usuario propietario de la función definida por el usuario que se va a cambiar. *owner_name* debe ser un Id. de usuario existente.

function_name

Es la función definida por el usuario que se va a cambiar. Los nombres de funciones deben seguir las reglas de los identificadores y deben ser únicos en la base de datos y para su propietario.

@parameter_name

Es un parámetro de la función definida por el usuario. Es posible declarar uno o varios parámetros. Una función puede tener un máximo de 1.024 parámetros. El usuario debe proporcionar el valor de cada parámetro declarado cuando se ejecuta la función (a menos que se defina un valor predeterminado para el parámetro). Cuando un parámetro de la función toma un valor predeterminado, debe especificarse la palabra clave "default" al llamar a la función para poder obtener el valor predeterminado. Este comportamiento es diferente del de los parámetros con valores predeterminados de los procedimientos almacenados, para los cuales omitir el parámetro implica especificar el valor predeterminado.

Especifique un nombre de parámetro con un signo (@) como el primer carácter. Los nombres de los parámetros deben cumplir las reglas de los identificadores. Los parámetros son locales para la función; los mismos nombres de parámetro se pueden utilizar en otras funciones. Los parámetros sólo pueden ocupar el lugar de constantes; no se pueden utilizar en lugar de nombres de tablas, nombres de columnas o nombres de otros objetos de base de datos.

scalar_parameter_data_type

Es el tipo de datos del parámetro. Todos los tipos de datos escalares, incluidos **bigint** y **sql_variant**, se pueden utilizar como un parámetro para las funciones definidas por el usuario. No se acepta el tipo de datos **timestamp**. No es posible especificar tipos no escalares como **cursor** y **table**.

scalar_return_data_type

Es el valor de retorno de una función escalar definida por el usuario. *scalar_return_data_type* puede ser de cualquiera de los tipos de datos escalares compatibles con SQL Server, excepto **text**, **ntext**, **image** y **timestamp**.

scalar_expression

Especifica que la función escalar devuelve un valor escalar.

TABLE

Especifica que el valor de retorno de la función que devuelve valores de tabla es una tabla.

En las funciones que devuelven valores de tabla en línea, el valor de retorno de TABLE se define mediante una única instrucción SELECT. Las funciones en línea no tienen asociadas variables de retorno.

En las funciones que devuelven valores de tabla de múltiples instrucciones, *@return_variable* es una variable TABLE, que se utiliza para almacenar y acumular las filas que se deben devolver como valor de la función.

function_body

Especifica que una serie de instrucciones Transact-SQL, que juntas no producen ningún efecto secundario, definen el valor de la función. *function_body* sólo se utiliza en funciones escalares y funciones de valores de tabla de múltiples instrucciones.

En las funciones escalares, *function_body* es una serie de instrucciones Transact-SQL que juntas dan como resultado un valor escalar.

En las funciones que devuelven valores de tabla de múltiples instrucciones, *function_body* es una serie de instrucciones Transact-SQL que rellena una variable de retorno de tabla.

select-stmt

Es la instrucción SELECT individual que define el valor de retorno de una función en línea que devuelve valores de tabla.

ENCRYPTION

Indica que SQL Server cifra las columnas de la tabla del sistema que contienen el texto de la instrucción CREATE FUNCTION. El uso de ENCRYPTION impide que la función se publique como parte de la duplicación de SQL Server.

SCHEMABINDING

Especifica que la función está enlazada a los objetos de base de datos a los que hace referencia. Esta condición impedirá cambios en la función si otros objetos enlazados del esquema hacen referencia a la misma.

El enlace de la función a los objetos a los que hace referencia sólo se elimina cuando se ejecuta una de estas dos acciones:

- Se quita la función.
- La función se cambia (mediante la instrucción ALTER) sin especificar la opción SCHEMABINDING.

Observaciones

No es posible utilizar ALTER FUNCTION para cambiar una función de valores escalares por una función de valores de tabla, ni viceversa. Tampoco es posible utilizar ALTER FUNCTION para cambiar una función en línea por una función de múltiples instrucciones o viceversa.

DROP FUNCTION

Quita una o más funciones definidas por el usuario de la base de datos actual. Las funciones definidas por el usuario se crean mediante CREATE FUNCTION y se modifican con ALTER FUNCTION.

Sintaxis

DROP FUNCTION { [*owner_name* **.**] *function_name* } [**,***...n*]

Argumentos

function_name

Es el nombre de la función definida por el usuario que se va a quitar. Especificar el nombre del propietario es opcional, no se pueden especificar el nombre de servidor ni el nombre de base de datos.

n

Es un marcador de posición que indica que se pueden especificar varias funciones definidas por el usuario.

INVOCACION DE FUNCIONES DEFINIDAS POR EL USUARIO

Cuando haga referencia o invoque a una función definida por el usuario, especifique el nombre de la función seguido de paréntesis. Dentro de los paréntesis, puede especificar expresiones llamadas argumentos que proporcionan los datos que se van a pasar a los parámetros. Al invocar a una función no puede especificar nombres de parámetros en los argumentos. Cuando invoque una función, debe proporcionar valores de argumentos para todos los parámetros y debe especificar los valores de argumentos en la misma secuencia en que están definidos los parámetros en la instrucción CREATE FUNCTION. Por ejemplo, si se define una función llamada fn_MyIntFunc que devuelve un entero con un parámetro entero y un parámetro **nchar(20)**, se puede invocar utilizando:

```
SELECT *
FROM SomeTable
```

```
WHERE PriKey = dbo.fn_MyIntFunc( 1, N'Anderson' )
```

Éste es un ejemplo de invocación de una función llamada fn_MyTableFunc definida para devolver una tabla:

```
SELECT *
FROM dbo.fn_MyTableFunc( 123.09, N'O''Neill' )
```

PROCEDIMIENTOS

CREATE PROCEDURE

Crea un procedimiento almacenado, que es una colección guardada de instrucciones Transact-SQL que puede tomar y devolver los parámetros proporcionados por el usuario.

Los procedimientos se pueden crear para uso permanente o para uso temporal en una sesión (procedimiento local temporal) o para su uso temporal en todas las sesiones (procedimiento temporal global).

Los procedimientos almacenados se pueden crear también para que se ejecuten automáticamente cuando se inicia Microsoft® SQL Server™.

Sintaxis

CREATE PROC [EDURE] *procedure_name* [**;** *number*]
[{ *@parameter data_type* }
[VARYING] [= *default*] [OUTPUT]
] [**,***...n*]

[WITH
{ RECOMPILE | ENCRYPTION | RECOMPILE **,** ENCRYPTION }]

[FOR REPLICATION]

AS *sql_statement* [*...n*]

Argumentos

procedure_name

Es el nombre del nuevo procedimiento almacenado. Los nombres de procedimiento deben seguir las reglas de los identificadores y deben ser únicos en la base de datos y para su propietario.

Los procedimientos temporales locales o globales se pueden crear precediendo el *procedure_name* con un signo numérico simple (#*procedure_name*) para los procedimientos temporales locales y un signo numérico doble (##*procedure_name*) para los procedimientos temporales globales. El nombre completo, incluidos los signos # o ##, no puede exceder de 128 caracteres. Especificar el nombre del propietario del procedimiento es opcional.

;*number*

Es un entero opcional utilizado para agrupar procedimientos del mismo nombre de forma que se puedan quitar juntos con una única instrucción DROP PROCEDURE. Por ejemplo, los procedimientos utilizados con una aplicación llamada "orders" se pueden llamar **orderproc**;1, **orderproc**;2, etc. La instrucción DROP PROCEDURE **orderproc** quita el grupo completo. Si el nombre contiene identificadores delimitados, el número no debe incluirse como parte del identificador; utilice el delimitador adecuado sólo alrededor de *procedure_name*.

@parameter

Es un parámetro del procedimiento. En una instrucción CREATE PROCEDURE se pueden declarar uno o más parámetros. El usuario debe proporcionar el valor de cada parámetro declarado cuando se ejecuta el procedimiento, a menos que se haya definido un valor predeterminado para el parámetro. Un procedimiento almacenado puede tener un máximo de 2.100 parámetros.

Especifique un nombre de parámetro con un signo (@) como el primer carácter. Los nombres de los parámetros deben cumplir las reglas de los identificadores. Los parámetros son locales para el procedimiento; los mismos nombres de parámetro se pueden utilizar en otros procedimientos. De forma predeterminada, los parámetros sólo pueden ocupar el lugar de constantes; no se pueden utilizar en lugar de nombres de tablas, nombres de columnas o nombres de otros objetos de base de datos.

data_type

Es el tipo de datos del parámetro. Todos los tipos de datos, incluidos **text**, **ntext** e **image**, se pueden utilizar como parámetros de un procedimiento almacenado. Sin embargo, el tipo de datos **cursor** sólo se puede utilizar en parámetros OUTPUT. Cuando se especifica un tipo de datos **cursor**, deben especificarse también las palabras clave VARYING y OUTPUT.

> **Nota** No hay límite para el número máximo de parámetros de salida que pueden ser del tipo de datos **cursor**.

VARYING

Especifica el conjunto de resultados admitido como parámetro de salida (generado dinámicamente por medio del procedimiento almacenado y cuyo contenido puede variar). Sólo se aplica a los parámetros de cursor.

default

Es un valor predeterminado para el parámetro. Si se define un valor predeterminado, el procedimiento se puede ejecutar sin especificar un valor para ese parámetro. El valor predeterminado debe ser una constante o puede ser NULL. Si el procedimiento utiliza el parámetro con la palabra clave LIKE, puede incluir caracteres comodín (%, _, [] y [^]).

OUTPUT

Indica que se trata de un parámetro de retorno. El valor de esta opción puede devolverse a EXEC[UTE]. Utilice los parámetros OUTPUT para devolver información al procedimiento que llama. Los parámetros **text**, **ntext** e **image** se pueden utilizar como parámetros OUTPUT. Un parámetro de salida que utilice la palabra clave OUTPUT puede ser un marcador de posición de cursor.

n

Es un marcador de posición que indica que se pueden especificar un máximo de 2.100 parámetros.

{RECOMPILE | ENCRYPTION | RECOMPILE**,** ENCRYPTION}

RECOMPILE indica que SQL Server no almacena en la caché un plan para este procedimiento y el procedimiento se vuelve a compilar en tiempo de ejecución. Utilice la opción RECOMPILE cuando utilice valores atípicos o temporales y no desee anular el plan de ejecución que está almacenado en la memoria caché.

ENCRYPTION indica que SQL Server codifica la entrada de la tabla **syscomments** que contiene el texto de la instrucción CREATE PROCEDURE. El uso de ENCRYPTION impide que el procedimiento se publique como parte de la duplicación de SQL Server.

> **Nota** Durante una actualización, SQL Server utiliza los comentarios cifrados almacenados en **syscomments** para volver a crear los procedimientos cifrados.

FOR REPLICATION

Especifica que los procedimientos almacenados creados para la duplicación no se pueden ejecutar en el suscriptor. Se utiliza un procedimiento almacenado creado con la opción FOR REPLICATION como un filtro de procedimiento almacenado y sólo se ejecuta durante la duplicación. Esta opción no se puede utilizar con la opción WITH RECOMPILE.

AS

Son las acciones que va a llevar a cabo el procedimiento.

sql_statement

Es cualquier número y tipo de instrucciones Transact-SQL que se incluirán en el procedimiento. Tiene algunas limitaciones.

n

Se trata de un marcador de posición que indica que se pueden incluir varias instrucciones Transact-SQL en el procedimiento.

Observaciones

El tamaño máximo de un procedimiento almacenado es 128 MB.

Un procedimiento almacenado definido por el usuario sólo puede crearse en la base de datos actual (excepto los procedimientos temporales, que se crean siempre en **tempdb**). La instrucción CREATE PROCEDURE no se puede combinar con otras instrucciones Transact-SQL en un único proceso por lotes.

De forma predeterminada, los parámetros pueden aceptar valores NULL. Si se pasa un valor de parámetro NULL y ese parámetro se utiliza en una instrucción CREATE o ALTER TABLE que hace referencia a una columna que no admite valores NULL, SQL Server genera un error. Para impedir que se pase un valor de parámetro NULL a una columna que no admite NULL, agregue código al procedimiento o utilice un valor predeterminado (con la palabra clave DEFAULT de CREATE o ALTER TABLE) para la columna.

Se recomienda que indique explícitamente NULL o NOT NULL para cada columna de las instrucciones CREATE TABLE o ALTER TABLE de un procedimiento almacenado, por ejemplo, cuando se crea una tabla temporal. Las opciones ANSI_DFLT_ON y ANSI_DFLT_OFF controlan la forma en la que SQL Server asigna los atributos NULL o NOT NULL a las columnas si no se especifican en una instrucción CREATE TABLE o ALTER TABLE. Si una conexión ejecuta un procedimiento almacenado con valores distintos para estas opciones de los que utilizó la conexión que creó el procedimiento, las columnas de la tabla creada para la segunda conexión pueden tener distintas opciones de admitir valores NULL y exhibir distintos comportamientos. Si se especifica NULL o NOT NULL explícitamente para cada columna, las tablas temporales se crean con la misma capacidad de admitir valores NULL para todas las conexiones que ejecutan el procedimiento almacenado.

SQL Server guarda los valores de SET QUOTED_IDENTIFIER y de SET ANSI_NULLS cuando se crea o altera un procedimiento almacenado. Estos valores originales se utilizan cuando se ejecuta el procedimiento almacenado. Por tanto, cualquier valor de sesión de cliente de SET QUOTED_IDENTIFIER y SET ANSI_NULLS se omitirá durante la ejecución del procedimiento almacenado. Las instrucciones SET QUOTED_IDENTIFIER y SET ANSI_NULLS que se producen en el procedimiento almacenado no afectan a la funcionalidad del mismo.

Otras opciones de SET, como SET ARITHABORT, SET ANSI_WARNINGS o SET ANSI_PADDINGS no se guardan cuando se crea o se altera un procedimiento almacenado. Si la lógica del procedimiento almacenado depende de un valor específico, incluya una instrucción SET al inicio del procedimiento para asegurar el valor adecuado. Cuando una instrucción SET se ejecuta desde un procedimiento almacenado, el valor permanece en efecto sólo hasta que se completa el procedimiento almacenado. A continuación, la configuración vuelve al valor que tenía cuando se llamó al procedimiento almacenado. Esto

permite a los clientes individuales establecer las opciones deseadas sin afectar a la lógica del procedimiento almacenado.

> **Nota** El hecho de que SQL Server interprete una cadena vacía como un espacio individual o como una verdadera cadena vacía se controla mediante el valor de nivel de compatibilidad. Si el nivel de compatibilidad es menor o igual que 65, SQL Server interpreta las cadenas vacías como espacios simples. Si el nivel de compatibilidad es igual a 70, SQL Server interpreta las cadenas vacías como tales.

Obtener información acerca de procedimientos almacenados

Para mostrar el texto utilizado para crear el procedimiento, ejecute **sp_helptext** en la base de datos en que existe el procedimiento, con el nombre del procedimiento como parámetro.

> **Nota** Los procedimientos almacenados creados con la opción ENCRYPTION no pueden verse con **sp_helptext**.

Para cambiar el nombre de un procedimiento, utilice **sp_rename**.

Referencia a objetos

SQL Server permite crear procedimientos almacenados que hacen referencia a objetos que todavía no existen. En el momento de la creación, sólo se realiza la comprobación de sintaxis. El procedimiento almacenado se compila para generar un plan de ejecución cuando se ejecute, si no existe ya un plan válido en la caché. Solamente durante la compilación se resuelven todos los objetos a los que se hace referencia en el procedimiento almacenado. Por tanto, se puede crear correctamente un procedimiento almacenado sintácticamente correcto que hace referencia a objetos que todavía no existen, aunque provocará un error en el momento de la ejecución, porque los objetos a los que hace referencia no existen.

Resolución diferida de nombres y nivel de compatibilidad

SQL Server permite que los procedimientos almacenados Transact-SQL hagan referencia a tablas que no existen en el momento de la creación. Esta capacidad se denomina resolución diferida de nombres. No obstante, si el procedimiento almacenado Transact-SQL hace referencia a una tabla definida en el procedimiento almacenado, se emite una advertencia en tiempo de creación si el nivel de compatibilidad (establecido al ejecutar **sp_dbcmptlevel**) es 65. Si la tabla a la que se hace referencia no existe, se devuelve un mensaje de error en tiempo de ejecución.

ALTER PROCEDURE

Modifica un procedimiento creado anteriormente por la ejecución de la instrucción CREATE PROCEDURE, sin cambiar los permisos y sin que afecte a ningún procedimiento almacenado ni desencadenador dependientes.

Sintaxis

```
ALTER PROC [ EDURE ] procedure_name [ ; number ]
   [ { @parameter data_type }
      [ VARYING ] [ = default ] [ OUTPUT ]
   ] [ ,...n ]
[ WITH
   { RECOMPILE | ENCRYPTION
      | RECOMPILE , ENCRYPTION
   }
]
[ FOR REPLICATION ]
AS
   sql_statement [ ...n ]
```

Argumentos

procedure_name

Es el nombre del procedimiento que se va a cambiar. Los nombres de los procedimientos deben seguir las reglas para los identificadores.

;*number*

Es un entero opcional existente que se utiliza para agrupar los procedimientos del mismo nombre, de forma que puedan quitarse juntos con una sola instrucción DROP PROCEDURE.

@parameter

Es un parámetro del procedimiento.

data_type

Es el tipo de datos del parámetro.

VARYING

Especifica el conjunto de resultados admitido como parámetro de salida (generado dinámicamente por medio del procedimiento almacenado y cuyo contenido puede variar). Sólo se aplica a los parámetros de cursor.

default

Es un valor predeterminado para el parámetro.

OUTPUT

Indica que se trata de un parámetro de retorno.

n

Es un marcador de posición que indica que se pueden especificar hasta 2.100 parámetros.

{RECOMPILE | ENCRYPTION | RECOMPILE**,** ENCRYPTION}

RECOMPILE indica que Microsoft® SQL Server™ no almacena en la memoria caché un plan para este procedimiento, por lo que el procedimiento se vuelve a compilar en tiempo de ejecución.

ENCRYPTION indica que SQL Server cifra la entrada de la tabla **syscomments** que contiene el texto de la instrucción ALTER PROCEDURE. El uso de ENCRYPTION impide que el procedimiento se publique como parte de la duplicación de SQL Server.

> **Nota** Durante una actualización, SQL Server utiliza los comentarios cifrados almacenados en **syscomments** para volver a crear los procedimientos cifrados.

FOR REPLICATION

Especifica que los procedimientos almacenados creados para la duplicación no se pueden ejecutar en el suscriptor. Se utiliza un procedimiento almacenado creado con la opción FOR REPLICATION como un filtro de procedimiento almacenado y sólo se ejecuta durante la duplicación. Esta opción no se puede utilizar con la opción WITH RECOMPILE.

AS

Son las acciones que va a llevar a cabo el procedimiento.

sql_statement

Es cualquier número y tipo de instrucciones Transact-SQL que se incluirán en el procedimiento. Tiene algunas limitaciones.

n

Es un marcador de posición que indica que se pueden incluir varias instrucciones Transact-SQL en el procedimiento.

> **Nota** Si anteriormente se creó una definición de procedimiento mediante WITH ENCRYPTION o WITH RECOMPILE, estas opciones sólo se activan si se incluyen en ALTER PROCEDURE.

Ejemplos

El ejemplo siguiente crea un procedimiento denominado **Oakland_authors** que, de forma predeterminada, contiene todos los autores de la ciudad de Oakland, California. Primero se conceden los permisos. A continuación, cuando el procedimiento debe cambiarse para obtener todos los autores de California, se utiliza ALTER PROCEDURE para volver a definir el procedimiento almacenado.

```
USE pubs
GO
IF EXISTS(SELECT name FROM sysobjects WHERE name = 'Oakland_authors' AND
type = 'P')
   DROP PROCEDURE Oakland_authors
GO
-- Create a procedure from the authors table that contains author
-- information for those authors who live in Oakland, California.
USE pubs
GO
CREATE PROCEDURE Oakland_authors
AS
SELECT au_fname, au_lname, address, city, zip
FROM pubs..authors
WHERE city = 'Oakland'
and state = 'CA'
ORDER BY au_lname, au_fname
GO
-- Here is the statement to actually see the text of the procedure.
SELECT o.id, c.text
FROM sysobjects o INNER JOIN syscomments c ON o.id = c.id
WHERE o.type = 'P' and o.name = 'Oakland_authors'
-- Here, EXECUTE permissions are granted on the procedure to public.
GRANT EXECUTE ON Oakland_authors TO public
GO
-- The procedure must be changed to include all
-- authors from California, regardless of what city they live in.
-- If ALTER PROCEDURE is not used but the procedure is dropped
-- and then re-created, the above GRANT statement and any
-- other statements dealing with permissions that pertain to this
-- procedure must be re-entered.
ALTER PROCEDURE Oakland_authors
```

```
WITH ENCRYPTION
AS
SELECT au_fname, au_lname, address, city, zip
FROM pubs..authors
WHERE state = 'CA'
ORDER BY au_lname, au_fname
GO
-- Here is the statement to actually see the text of the procedure.
SELECT o.id, c.text
FROM sysobjects o INNER JOIN syscomments c ON o.id = c.id
WHERE o.type = 'P' and o.name = 'Oakland_authors'
GO
```

DROP PROCEDURE

Quita uno o más procedimientos almacenados o grupos de procedimientos de la base de datos actual.

Sintaxis

DROP PROCEDURE { *procedure* } [**,**...*n*]

Argumentos

procedure

Es el nombre del procedimiento almacenado o grupo de procedimientos almacenados que se va a eliminar. Los nombres de los procedimientos deben seguir las reglas para los identificadores. Especificar el nombre del propietario del procedimiento es opcional y no se puede especificar un nombre de servidor o de base de datos.

n

Es un marcador de posición que indica que se pueden especificar varios procedimientos.

Ejemplos

Este ejemplo quita el procedimiento almacenado **byroyalty** (de la base de datos actual).

```
DROP PROCEDURE byroyalty
GO
```

EJECUTAR PROCEDIMIENTOS ALMACENADOS

Cuando una instrucción CREATE PROCEDURE se ejecuta correctamente, el nombre del procedimiento se almacena en la tabla del sistema **sysobjects** y el texto de la instrucción CREATE PROCEDURE se almacena en **syscomments**. Cuando se ejecuta por primera vez, el procedimiento se compila para determinar que dispone de un plan de acceso óptimo para recuperar los datos.

Parámetros con el tipo de datos cursor

Los procedimientos almacenados sólo pueden utilizar el tipo de datos **cursor** para los parámetros OUTPUT. Si se especifica el tipo de datos **cursor** para un parámetro, se requieren los parámetros VARYING y OUTPUT. Si se especifica la palabra clave VARYING para un parámetro, el tipo de datos debe ser **cursor** y se debe especificar la palabra clave OUTPUT.

Parámetros de salida cursor

Las siguientes reglas se aplican a los parámetros de salida **cursor** cuando se ejecuta el procedimiento:

- Para un cursor sólo hacia adelante, las filas devueltas en el conjunto de resultados del cursor son sólo aquellas filas que estén en la posición del cursor y hacia adelante al concluir la ejecución del procedimiento almacenado, por ejemplo:

    - En un procedimiento de un conjunto de resultados llamado RS de 100 filas, se abre un cursor no desplazable.
    - El procedimiento busca las primeras 5 filas del conjunto de resultados RS.
    - El procedimiento vuelve a quien le llamó.
    - El conjunto de resultados que RS devolvió a quien llamó consta de las filas 6 a 100 de RS y el cursor del que llama se coloca antes de la primera fila de RS.

- Para un cursor sólo hacia adelante, si el cursor se coloca antes de la primera fila una vez que finalice el procedimiento almacenado, el conjunto de resultados completo se devuelve al proceso por lotes, procedimiento almacenado o desencadenador que

llamó. Cuando se devuelve, la posición del cursor se establece antes de la primera fila.

- Para un cursor sólo hacia adelante, si el cursor se coloca después del final de la última fila una vez que finalice el procedimiento almacenado, se devolverá un conjunto de resultados vacío al proceso por lotes, procedimiento almacenado o desencadenador que llamó.

Nota Un conjunto de resultados vacío no es lo mismo que un valor NULL.

- Para un cursor desplazable, todas las filas del conjunto de resultados se devuelven al proceso por lotes, procedimiento almacenado o desencadenador que llama cuando finaliza la ejecución del procedimiento almacenado. Cuando se devuelve, la posición del cursor se deja en la posición de la última recuperación de datos ejecutada en el procedimiento.

- Para cualquier tipo de cursor, si se ha cerrado el cursor, se devuelve un valor NULL al proceso por lotes, procedimiento almacenado o desencadenador que llamó. Esto también ocurrirá si se ha asignado un cursor a un parámetro, pero ese cursor nunca se abre.

Nota El estado cerrado sólo tiene importancia en el momento del retorno. Por ejemplo, es válido cerrar un cursor a mitad del procedimiento, abrirlo posteriormente en el procedimiento y devolver el conjunto de resultados de ese cursor al proceso por lotes, procedimiento almacenado o desencadenador que llamó.

Procedimientos almacenados temporales

SQL Server admite dos tipos de procedimientos temporales: locales y globales. Un procedimiento temporal local sólo es visible para la conexión que lo creó. Un procedimiento temporal global está disponible para todas las conexiones. Los procedimientos temporales locales se quitan automáticamente al final de la sesión actual. Los procedimientos temporales globales se quitan al final de la última sesión que utiliza el procedimiento. Normalmente, esto sucede cuando finaliza la sesión que creó el procedimiento.

Los procedimientos temporales con nombres que incluyen # y ## pueden ser creados por cualquier usuario. Una vez creado el procedimiento, el propietario del procedimiento local es el único que puede utilizarlo. El permiso para ejecutar un procedimiento local temporal no se puede conceder a otros usuarios. Si se crea un procedimiento temporal global, todos los usuarios tienen acceso al mismo; los permisos no se pueden revocar explícitamente. Sólo aquéllos que cuenten con permiso CREATE PROCEDURE explícito en la base de datos **tempdb** pueden crear explícitamente un procedimiento temporal en **tempdb** (nombre

sin signo numérico). Los permisos se pueden conceder y revocar desde estos procedimientos.

Anidamiento de procedimientos almacenados

Los procedimientos almacenados pueden anidarse, es decir, un procedimiento almacenado llama a otro. El nivel de anidamiento aumenta cuando el procedimiento llamado inicia la ejecución y disminuye cuando el procedimiento llamado finaliza la ejecución. Sobrepasar los niveles máximos de anidamiento hace que la cadena completa de los procedimientos de llamada cause un error. La función @@NESTLEVEL devuelve el nivel actual de anidamiento.

Ejemplos

A. Utilizar un procedimiento sencillo con una instrucción SELECT compleja

Este procedimiento almacenado devuelve todos los autores (nombre y apellidos), los títulos y los publicadores a partir de una combinación de cuatro tablas. Este procedimiento almacenado no utiliza ningún parámetro.

```
USE pubs
IF EXISTS (SELECT name FROM sysobjects
         WHERE name = 'au_info_all' AND type = 'P')
   DROP PROCEDURE au_info_all
GO
CREATE PROCEDURE au_info_all
AS
SELECT au_lname, au_fname, title, pub_name
   FROM authors a INNER JOIN titleauthor ta
      ON a.au_id = ta.au_id INNER JOIN titles t
      ON t.title_id = ta.title_id INNER JOIN publishers p
      ON t.pub_id = p.pub_id
GO
```

El procedimiento almacenado **au_info_all** se puede ejecutar de estas formas:

```
EXECUTE au_info_all
-- Or
EXEC au_info_all
```

O si este procedimiento es la primera instrucción del proceso por lotes:

```
au_info_all
```

B. Utilizar un procedimiento sencillo con parámetros

Este procedimiento almacenado devuelve sólo los autores especificados (nombre y apellidos), los títulos y los publicadores a partir de una combinación de cuatro tablas. Este procedimiento almacenado acepta coincidencias exactas de los parámetros pasados.

```
USE pubs
IF EXISTS (SELECT name FROM sysobjects
         WHERE name = 'au_info' AND type = 'P')
   DROP PROCEDURE au_info
GO
USE pubs
GO
CREATE PROCEDURE au_info
   @lastname varchar(40),
   @firstname varchar(20)
AS
SELECT au_lname, au_fname, title, pub_name
   FROM authors a INNER JOIN titleauthor ta
      ON a.au_id = ta.au_id INNER JOIN titles t
      ON t.title_id = ta.title_id INNER JOIN publishers p
      ON t.pub_id = p.pub_id
   WHERE  au_fname = @firstname
      AND au_lname = @lastname
GO
```

El procedimiento almacenado **au_info** se puede ejecutar de estas formas:

```
EXECUTE au_info 'Dull', 'Ann'
-- Or
EXECUTE au_info @lastname = 'Dull', @firstname = 'Ann'
-- Or
EXECUTE au_info @firstname = 'Ann', @lastname = 'Dull'
-- Or
EXEC au_info 'Dull', 'Ann'
-- Or
EXEC au_info @lastname = 'Dull', @firstname = 'Ann'
-- Or
EXEC au_info @firstname = 'Ann', @lastname = 'Dull'
```

O si este procedimiento es la primera instrucción del proceso por lotes:

```
au_info 'Dull', 'Ann'
-- Or
au_info @lastname = 'Dull', @firstname = 'Ann'
-- Or
au_info @firstname = 'Ann', @lastname = 'Dull'
```

C. Utilizar un procedimiento sencillo con parámetros comodín

Este procedimiento almacenado devuelve sólo los autores especificados (nombre y apellidos), los títulos y los publicadores a partir de una combinación de cuatro tablas. Este patrón de procedimiento almacenado coincide con los parámetros pasados o si éstos no se proporcionan, utiliza los valores predeterminados.

```
USE pubs
IF EXISTS (SELECT name FROM sysobjects
      WHERE name = 'au_info2' AND type = 'P')
   DROP PROCEDURE au_info2
GO
USE pubs
GO
CREATE PROCEDURE au_info2
   @lastname varchar(30) = 'D%',
   @firstname varchar(18) = '%'
AS
SELECT au_lname, au_fname, title, pub_name
FROM authors a INNER JOIN titleauthor ta
   ON a.au_id = ta.au_id INNER JOIN titles t
   ON t.title_id = ta.title_id INNER JOIN publishers p
   ON t.pub_id = p.pub_id
WHERE au_fname LIKE @firstname
   AND au_lname LIKE @lastname
GO
```

El procedimiento almacenado **au_info2** se puede ejecutar en muchas combinaciones. Aquí se muestran sólo algunas combinaciones:

```
EXECUTE au_info2
-- Or
EXECUTE au_info2 'Wh%'
-- Or
EXECUTE au_info2 @firstname = 'A%'
-- Or
EXECUTE au_info2 '[CK]ars[OE]n'
-- Or
EXECUTE au_info2 'Hunter', 'Sheryl'
-- Or
EXECUTE au_info2 'H%', 'S%'
```

D. Utilizar parámetros OUTPUT

Los parámetros OUTPUT permiten a un procedimiento externo, un proceso por lotes o más de una instrucción Transact-SQL tener acceso a un conjunto de valores durante la ejecución del procedimiento. En el ejemplo siguiente, se crea un procedimiento almacenado (**titles_sum**) que admite un parámetro opcional de entrada y un parámetro de salida.

Primero, cree el procedimiento:

```
USE pubs
GO
IF EXISTS(SELECT name FROM sysobjects
      WHERE name = 'titles_sum' AND type = 'P')
   DROP PROCEDURE titles_sum
GO
USE pubs
GO
CREATE PROCEDURE titles_sum @@TITLE varchar(40) = '%', @@SUM money OUTPUT
AS
```

```
SELECT 'Title Name' = title
FROM titles
WHERE title LIKE @@TITLE
SELECT @@SUM = SUM(price)
FROM titles
WHERE title LIKE @@TITLE
GO
```

A continuación, utilice el parámetro OUTPUT con lenguaje de control de flujo.

> **Nota** La variable OUTPUT debe definirse durante la creación de la tabla, así como durante la utilización de la variable.

El nombre del parámetro y de la variable no tienen por qué coincidir; sin embargo, el tipo de datos y la posición de los parámetros deben coincidir (a menos que se utilice **@@**SUM = *variable*).

```
DECLARE @@TOTALCOST money
EXECUTE titles_sum 'The%', @@TOTALCOST OUTPUT
IF @@TOTALCOST < 200
BEGIN
   PRINT ' '
   PRINT 'All of these titles can be purchased for less than $200.'
END
ELSE
   SELECT 'The total cost of these titles is $'
         + RTRIM(CAST(@@TOTALCOST AS varchar(20)))
```

El siguiente es el conjunto de resultados:

```
Title Name
--------------------------------------------------------------------------
The Busy Executive's Database Guide
The Gourmet Microwave
The Psychology of Computer Cooking

(3 row(s) affected)

Warning, null value eliminated from aggregate.

All of these titles can be purchased for less than $200.
```

E. Utilizar un parámetro OUTPUT cursor

Los parámetros OUTPUT **cursor** se utilizan para volver a pasar un cursor que sea local a un procedimiento almacenado, al proceso por lotes, procedimiento almacenado o desencadenador que llamó.

Primero, crea el procedimiento que declara y, a continuación, abre un cursor en la tabla de títulos:

```
USE pubs
IF EXISTS (SELECT name FROM sysobjects
      WHERE name = 'titles_cursor' and type = 'P')
DROP PROCEDURE titles_cursor
GO
CREATE PROCEDURE titles_cursor @titles_cursor CURSOR VARYING OUTPUT
AS
SET @titles_cursor = CURSOR
FORWARD_ONLY STATIC FOR
SELECT *
FROM titles

OPEN @titles_cursor
GO
```

A continuación, se ejecuta un proceso por lotes que declara una variable local de cursor, ejecuta el procedimiento para asignar el cursor a la variable local y, por último, busca las filas desde el cursor.

```
USE pubs
GO
DECLARE @MyCursor CURSOR
EXEC titles_cursor @titles_cursor = @MyCursor OUTPUT
WHILE (@@FETCH_STATUS = 0)
BEGIN
   FETCH NEXT FROM @MyCursor
END
CLOSE @MyCursor
DEALLOCATE @MyCursor
GO
```

F. Utilizar la opción WITH RECOMPILE

La cláusula WITH RECOMPILE es útil cuando los parámetros suministrados al procedimiento no son los típicos y cuando no debe almacenarse un nuevo plan de ejecución en la caché o en memoria.

```
USE pubs
IF EXISTS (SELECT name FROM sysobjects
      WHERE name = 'titles_by_author' AND type = 'P')
   DROP PROCEDURE titles_by_author
GO
CREATE PROCEDURE titles_by_author @@LNAME_PATTERN varchar(30) = '%'
WITH RECOMPILE
AS
SELECT RTRIM(au_fname) + ' ' + RTRIM(au_lname) AS 'Authors full name',
   title AS Title
FROM authors a INNER JOIN titleauthor ta
   ON a.au_id = ta.au_id INNER JOIN titles t
   ON ta.title_id = t.title_id
WHERE au_lname LIKE @@LNAME_PATTERN
GO
```

G. Utilizar la opción WITH ENCRYPTION

La cláusula WITH ENCRYPTION oculta a los usuarios el texto de un procedimiento almacenado. Este ejemplo crea un procedimiento cifrado, utiliza el procedimiento almacenado de sistema **sp_helptext** para obtener información sobre ese procedimiento y, a continuación, intenta obtener información sobre el procedimiento directamente de la tabla **syscomments**.

```
IF EXISTS (SELECT name FROM sysobjects
      WHERE name = 'encrypt_this' AND type = 'P')
   DROP PROCEDURE encrypt_this
GO
USE pubs
GO
CREATE PROCEDURE encrypt_this
WITH ENCRYPTION
AS
SELECT *
FROM authors
GO

EXEC sp_helptext encrypt_this
```

El siguiente es el conjunto de resultados:

```
The object's comments have been encrypted.
```

A continuación, seleccione el número de identificación y el texto del contenido del procedimiento almacenado cifrado.

```
SELECT c.id, c.text
FROM syscomments c INNER JOIN sysobjects o
   ON c.id = o.id
WHERE o.name = 'encrypt_this'
```

El siguiente es el conjunto de resultados:

> **Nota** El resultado de la columna **text** se muestra en una línea separada. Cuando se ejecuta, esta información aparece en la misma línea que la información de la columna **id**.

```
id         text
---------- ------------------------------------------------------------
1413580074
????????????????????????????????e????????????????????????????????????
????????????????????????????????

(1 row(s) affected)
```

H. Crear un procedimiento almacenado del sistema definido por el usuario

El ejemplo siguiente crea un procedimiento para mostrar todas las tablas y sus índices correspondientes con un nombre de tabla que empieza con la cadena **emp**. Si no se especifica, este procedimiento devuelve todas las tablas (e índices) con un nombre de tabla que empieza con **sys**.

```
IF EXISTS (SELECT name FROM sysobjects
      WHERE name = 'sp_showindexes' AND type = 'P')
   DROP PROCEDURE sp_showindexes
GO
USE master
GO
CREATE PROCEDURE sp_showindexes
   @@TABLE varchar(30) = 'sys%'
AS
SELECT o.name AS TABLE_NAME,
   i.name AS INDEX_NAME,
   indid AS INDEX_ID
FROM sysindexes i INNER JOIN sysobjects o
   ON o.id = i.id
WHERE o.name LIKE @@TABLE
GO
USE pubs
EXEC sp_showindexes 'emp%'
GO
```

El siguiente es el conjunto de resultados:

```
TABLE_NAME       INDEX_NAME       INDEX_ID
---------------- ---------------- ----------------
employee         employee_ind     1
employee         PK_emp_id        2

(2 row(s) affected)
```

I. Utilizar la resolución diferida de nombres

Este ejemplo muestra cuatro procedimientos y las distintas formas en que se puede utilizar la resolución diferida de nombres. Cada procedimiento almacenado se crea aunque la tabla o columna a la que hace referencia no exista en el momento de la compilación.

```
IF EXISTS (SELECT name FROM sysobjects
      WHERE name = 'proc1' AND type = 'P')
   DROP PROCEDURE proc1
GO
-- Creating a procedure on a nonexistent table.
USE pubs
GO
CREATE PROCEDURE proc1
AS
   SELECT *
   FROM does_not_exist
GO
-- Here is the statement to actually see the text of the procedure.
SELECT o.id, c.text
```

```
FROM sysobjects o INNER JOIN syscomments c
   ON o.id = c.id
WHERE o.type = 'P' AND o.name = 'proc1'
GO
USE master
GO
IF EXISTS (SELECT name FROM sysobjects
      WHERE name = 'proc2' AND type = 'P')
   DROP PROCEDURE proc2
GO
-- Creating a procedure that attempts to retrieve information from a
-- nonexistent column in an existing table.
USE pubs
GO
CREATE PROCEDURE proc2
AS
   DECLARE @middle_init char(1)
   SET @middle_init = NULL
   SELECT au_id, middle_initial = @middle_init
   FROM authors
GO
-- Here is the statement to actually see the text of the procedure.
SELECT o.id, c.text
FROM sysobjects o INNER JOIN syscomments c
   ON o.id = c.id
WHERE o.type = 'P' and o.name = 'proc2'
```

CURSORES

Las operaciones de una base de datos relacional actúan en un conjunto completo de filas. El conjunto de filas que devuelve una instrucción SELECT está compuesto de todas las filas que satisfacen las condiciones de la cláusula WHERE de la instrucción. Este conjunto completo de filas que devuelve la instrucción se conoce como el conjunto de resultados. Las aplicaciones, especialmente las aplicaciones interactivas en línea, no siempre pueden trabajar de forma efectiva con el conjunto de resultados completo si lo toman como una unidad. Estas aplicaciones necesitan un mecanismo que trabaje con una fila o un pequeño bloque de filas cada vez. Los cursores son una extensión de los conjuntos de resultados que proporcionan dicho mecanismo.

Los cursores amplían el procesamiento de los resultados porque:

- Permiten situarse en filas específicas del conjunto de resultados.
- Recuperan una fila o bloque de filas de la posición actual en el conjunto de resultados.
- Aceptan modificaciones de los datos de las filas en la posición actual del conjunto de resultados
- Aceptan diferentes grados de visibilidad para los cambios que realizan otros usuarios en la información de la base de datos que se presenta en el conjunto de resultados.
- Proporcionan instrucciones de Transact-SQL en secuencias de comandos, procedimientos almacenados y acceso de desencadenadores a los datos de un conjunto de resultados.

Pedir un cursor

Microsoft® SQL Server™ 2000 admite el uso de dos métodos distintos para solicitar un cursor:

- Transact-SQL

  El lenguaje Transact-SQL admite una sintaxis para utilizar cursores modelados a partir de la sintaxis de cursores de SQL-92.

- Funciones de cursores para interfaces de programación de aplicaciones (API) de bases de datos.

  SQL Server admite la funcionalidad de cursores de las siguientes API de bases de datos:

- ADO (Microsoft ActiveX® Data Object)
- OLE DB
- ODBC (Conectividad abierta de bases de datos)
- DB-Library

Una aplicación nunca debe mezclar estos dos métodos de petición de cursores. Una aplicación que ya ha utilizado la API para determinar el comportamiento de cursores no debe ejecutar una instrucción DECLARE CURSOR de Transact-SQL para pedir también un cursor de Transact-SQL. Una aplicación sólo debería ejecutar DECLARE CURSOR si antes ha vuelto a fijar todos los atributos de cursores de la API a sus valores predeterminados.

Proceso de cursores

Si bien los cursores de Transact-SQL y los cursores de la API poseen una sintaxis diferente, se utiliza el siguiente proceso general con todos los cursores de SQL Server:

1. Asigne un cursor al conjunto de resultados de una instrucción Transact-SQL y defina las características del cursor como, por ejemplo, si sus filas se pueden actualizar.

2. Ejecute la instrucción de Transact-SQL para llenar el cursor.

3. Recupere las filas del cursor que desea ver. La operación de recuperar una fila o un bloque de filas de un cursor recibe el nombre de recopilación. Realizar series de recopilaciones para recuperar filas, ya sea hacia adelante o hacia atrás, recibe el nombre de desplazamiento.

4. Existe la opción de realizar operaciones de modificación (actualización o eliminación) en la fila de la posición actual del cursor.

5. Cierre el cursor.

@@CURSOR_ROWS

Devuelve el número de filas correspondientes actualmente al último cursor abierto en la conexión. Para mejorar el rendimiento, Microsoft® SQL Server™ puede llenar asincrónicamente los conjuntos de claves y los cursores estáticos de gran tamaño. Puede llamar a @@CURSOR_ROWS para determinar que el número de filas que cumplan las condiciones del cursor se recuperen en el momento en que se llama a @@CURSOR_ROWS.

Valor de retorno	Descripción
-m	El cursor se llena de forma asincrónica. El valor devuelto (*-m*) es el número de filas que contiene actualmente el conjunto de claves.
-1	El cursor es dinámico. Como los cursores dinámicos reflejan todos los cambios, el número de filas correspondientes al cursor cambia constantemente. Nunca se puede afirmar que se han recuperado todas las filas que correspondan.
0	No se han abierto cursores, no hay filas calificadas para el último cursor abierto, o éste se ha cerrado o su asignación se ha cancelado.
n	El cursor está completamente lleno. El valor obtenido (*n*) es el número total de filas del cursor.

Sintaxis

@@CURSOR_ROWS

Tipos devueltos

integer

Observaciones

El número que devuelve @@CURSOR_ROWS es negativo cuando el último cursor se ha abierto de forma asincrónica. Los cursores controlados por conjunto de claves y los estáticos se abren de forma asincrónica cuando el valor de **sp_configure cursor threshold** es mayor que cero y el número de filas del conjunto de resultados del cursor es mayor que su umbral.

Ejemplos

En este ejemplo se declara un cursor y se utiliza SELECT para mostrar el valor de @@CURSOR_ROWS. La opción tiene el valor 0 antes de abrir el cursor; el valor -1 indica que el conjunto de claves del cursor se está llenando de forma asincrónica.

```
SELECT @@CURSOR_ROWS
DECLARE authors_cursor CURSOR FOR
SELECT au_lname FROM authors
OPEN authors_cursor
FETCH NEXT FROM authors_cursor
SELECT @@CURSOR_ROWS
CLOSE authors_cursor
DEALLOCATE authors_cursor

-----------
0
```

```
(1 row(s) affected)

au_lname
----------------------------------------
White

(1 row(s) affected)


-----------
-1

(1 row(s) affected)
```

CURSOR_STATUS

Una función escalar que permite al que llama a un procedimiento almacenado determinar si el procedimiento ha devuelto un cursor y el conjunto de resultados de un determinado parámetro.

Sintaxis

CURSOR_STATUS
(
{ 'local' **,** '*cursor_name*' }
| { 'global' **,** '*cursor_name*' }
| { 'variable' **,** '*cursor_variable*' }
)

Argumentos

'local'

Especifica una constante que indica que el origen del cursor es un nombre local de cursor.

'*cursor_name*'

Es el nombre del cursor. Un nombre de cursor debe ajustarse a las reglas para los identificadores.

'global'

Especifica una constante que indica que el origen del cursor es un nombre global de cursor.

'variable'

Especifica una constante que indica que el origen del cursor es una variable local.

'*cursor_variable*'

Es el nombre de la variable de cursor. Una variable de cursor debe definirse mediante el tipo de datos **cursor**.

Tipos devueltos

smallint

Valor de retorno	**Nombre de cursor**	**Variable de cursor**
1	El conjunto de resultados del cursor tiene al menos una fila y: Para los cursores de conjuntos de claves y que no distinguen, el conjunto de resultados tiene al menos una fila. Para los cursores dinámicos, el conjunto de resultados puede tener cero, una o más filas.	El cursor asignado a esta variable está abierto y: Para los cursores de conjuntos de claves y que no distinguen, el conjunto de resultados tiene al menos una fila. Para los cursores dinámicos, el conjunto de resultados puede tener cero, una o más filas.
0	El conjunto de resultados del cursor está vacío.*	El cursor asignado a esta variable está abierto, pero el conjunto de resultados está definitivamente vacío.*
-1	El cursor está cerrado.	El cursor asignado a esta variable está cerrado.
-2	No aplicable.	Puede ser: El procedimiento llamado anteriormente no ha asignado ningún cursor a esta variable de resultado. El procedimiento llamado anteriormente asignó un cursor a esta variable de resultado, pero se encontraba en un estado cerrado al terminar el procedimiento. Por tanto, se cancela la asignación del cursor y no se devuelve al procedimiento que hace la llamada. No hay cursor asignado a una variable declarada de cursor.

-3	No existe ningún cursor con el nombre indicado.	No existe una variable de cursor con el nombre indicado o si existe, no tiene todavía un cursor asignado.

* Los cursores dinámicos no devuelven nunca este resultado.

Ejemplos

El ejemplo siguiente crea un procedimiento denominado **lake_list** y utiliza el resultado de la ejecución de **lake_list** como comprobación de CURSOR_STATUS.

Nota El ejemplo siguiente depende de un procedimiento denominado **check_authority**, que no se ha creado.

```
USE pubs
IF EXISTS (SELECT name FROM sysobjects
      WHERE name = 'lake_list' AND type = 'P')
   DROP PROCEDURE lake_list
GO
CREATE PROCEDURE lake_list
   ( @region varchar(30),
     @size integer,
     @lake_list_cursor CURSOR VARYING OUTPUT )
AS
BEGIN
   DECLARE @ok SMALLINT
   EXECUTE check_authority @region, username, @ok OUTPUT
   IF @ok = 1
      BEGIN
      SET @lake_list_cursor =CURSOR LOCAL SCROLL FOR
         SELECT name, lat, long, size, boat_launch, cost
         FROM lake_inventory
         WHERE locale = @region AND area >= @size
         ORDER BY name
      OPEN @lake_list_cursor
      END
END
DECLARE @my_lakes_cursor CURSOR
DECLARE @my_region char(30)
SET @my_region = 'Northern Ontario'
EXECUTE lake_list @my_region, 500, @my_lakes_cursor OUTPUT
IF Cursor_Status('variable', '@my_lakes_cursor') <= 0
   BEGIN
   /* Some code to tell the user that there is no list of
   lakes for him/her */
   END
ELSE
   BEGIN
      FETCH @my_lakes_cursor INTO -- Destination here
      -- Continue with other code here.
END
```

@@FETCH_STATUS

Devuelve el estado de la última instrucción FETCH de cursor ejecutada sobre cualquier cursor que la conexión haya abierto.

Valor de retorno	Descripción
0	La instrucción FETCH se ejecutó correctamente.
-1	La instrucción FETCH ha finalizado con error o la fila estaba más allá del conjunto de resultados.
-2	Falta la fila recuperada.

Sintaxis

@@FETCH_STATUS

Tipos devueltos

integer

Observaciones

Al ser @@FETCH_STATUS global para todos los cursores de una conexión, debe usarse con cuidado. Después de ejecutar una instrucción FETCH, la comprobación de @@FETCH_STATUS se debe realizar antes de que se ejecute otra instrucción FETCH sobre otro cursor. El valor de @@FETCH_STATUS no está definido antes de producirse las recuperaciones en la conexión.

Por ejemplo, supongamos que un usuario ejecuta una instrucción FETCH sobre un cursor y a continuación llama a un procedimiento almacenado que abre y procesa los resultados de otro cursor. Cuando vuelve el control desde el procedimiento almacenado llamado, @@FETCH_STATUS reflejará la última instrucción FETCH ejecutada en el procedimiento almacenado, no la ejecutada antes de llamar al procedimiento.

Ejemplos

Este ejemplo utiliza @@FETCH_STATUS para controlar las actividades del cursor en un bucle WHILE.

```
DECLARE Employee_Cursor CURSOR FOR
SELECT LastName, FirstName FROM Northwind.dbo.Employees
OPEN Employee_Cursor
FETCH NEXT FROM Employee_Cursor
WHILE @@FETCH_STATUS = 0
```

```
BEGIN
   FETCH NEXT FROM Employee_Cursor
END
CLOSE Employee_Cursor
DEALLOCATE Employee_Cursor
```

FETCH

Obtiene una fila específica de un cursor Transact-SQL del servidor.

Sintaxis

FETCH
[[NEXT | PRIOR | FIRST | LAST
| ABSOLUTE { *n* | **@***nvar* }
| RELATIVE { *n* | **@***nvar* }
]
FROM
]
{ { [GLOBAL] *cursor_name* } | *@cursor_variable_name* }
[INTO **@***variable_name* [**,**...*n*]]

Argumentos

NEXT

Devuelve la fila de resultados que sigue inmediatamente a la fila actual y la fila devuelta pasa a ser la fila actual. Si FETCH NEXT es la primera recuperación que se ejecuta en un cursor, devuelve la primera fila del conjunto de resultados. NEXT es la opción predeterminada de recuperación de cursor.

PRIOR

Devuelve la fila de resultados inmediatamente anterior a la fila actual y la fila devuelta pasa a ser la fila actual. Si FETCH PRIOR es la primera recuperación que se ejecuta en un cursor, no se devuelve ninguna fila y el cursor queda posicionado antes de la primera fila.

FIRST

Devuelve la primera fila del cursor y la convierte en la fila actual.

LAST

Devuelve la última fila del cursor y la convierte en la fila actual.

ABSOLUTE {*n* | **@***nvar*}

Si *n* o *@nvar* es positivo, devuelve la fila *n* desde el principio del cursor y la convierte en la nueva fila actual. Si *n* o *@nvar* es negativo, devuelve la fila *n* desde el final del cursor y la convierte en la nueva fila actual. Si *n* o *@nvar* es 0, no se devuelve ninguna fila; *n* debe ser una constante entera y *@nvar* debe ser **smallint**, **tinyint** o **int**.

RELATIVE {*n* | **@***nvar*}

Si *n* o *@nvar* es positivo, devuelve la fila que está *n* filas a continuación de la fila actual y la convierte en la nueva fila actual. Si *n* o *@nvar* es negativo, devuelve la fila que está *n* filas antes de la fila actual y la convierte en la nueva fila actual. Si *n* o *@nvar* es 0, devuelve la fila actual. Si FETCH RELATIVE se especifica con *n* o *@nvar* establecidas a números negativos o 0 en la primera recuperación que se hace en un cursor, no se devuelve ninguna fila; *n* debe ser una constante entera y *@nvar* debe ser **smallint**, **tinyint** o **int**.

GLOBAL

Especifica que *cursor_name* hace referencia a un cursor global.

cursor_name

Es el nombre del cursor abierto desde el que se debe realizar la recuperación. Si existen un cursor global y otro local con *cursor_namer* como nombre, *cursor_name* hace referencia al cursor global si se especifica GLOBAL y al cursor local si no se especifica GLOBAL.

@cursor_variable_name

Es el nombre de una variable de cursor que hace referencia al cursor abierto en el que se va efectuar la recuperación.

INTO *@variable_name*[**,**...*n*]

Permite que los datos de las columnas de una búsqueda pasen a variables locales. Todas las variables de la lista, de izquierda a derecha, están asociadas a las columnas correspondientes del conjunto de resultados del cursor. El tipo de datos de cada variable tiene que coincidir o ser compatible con la conversión implícita del tipo de datos de la columna correspondiente del conjunto de resultados. El número de variables tiene que coincidir con el número de columnas de la lista seleccionada en el cursor.

Observaciones

Si no se especifica la opción SCROLL en una instrucción DECLARE CURSOR del estilo SQL-92, NEXT es la única opción admitida de FETCH. Si se especifica SCROLL en una instrucción DECLARE CURSOR del estilo SQL-92, se admiten todas las opciones de FETCH.

Cuando se utilizan las extensiones de cursor DECLARE de Transact-SQL, hay que aplicar estas reglas:

- Si se especifica FORWARD-ONLY o FAST_FORWARD, NEXT es la única opción admitida de FETCH.
- Si no se especifican DYNAMIC, FORWARD_ONLY o FAST_FORWARD, y se especifican KEYSET, STATIC o SCROLL, se admiten todas las opciones de FETCH.
- Los cursores DYNAMIC SCROLL admiten todas las opciones de FETCH excepto ABSOLUTE.

La función @@FETCH_STATUS informa del estado de la última instrucción FETCH. La misma información queda grabada en la columna **fetch_status** del cursor devuelto por **sp_describe_cursor**. Esta información de estado se debe utilizar para determinar la validez de los datos devueltos por una instrucción FETCH antes de iniciar cualquier operación con los datos.

Ejemplos

A. Utilizar FETCH en un cursor sencillo

Este ejemplo declara un cursor sencillo para las filas de la tabla **authors** cuyo apellido empiece por B y utiliza FETCH NEXT para recorrer las filas. Las instrucciones FETCH devuelven el valor de la columna especificada en DECLARE CURSOR como conjunto de resultados de una sola fila.

```
USE pubs
GO
DECLARE authors_cursor CURSOR FOR
SELECT au_lname FROM authors
WHERE au_lname LIKE "B%"
ORDER BY au_lname

OPEN authors_cursor

-- Perform the first fetch.
FETCH NEXT FROM authors_cursor

-- Check @@FETCH_STATUS to see if there are any more rows to fetch.
WHILE @@FETCH_STATUS = 0
BEGIN
   -- This is executed as long as the previous fetch succeeds.
   FETCH NEXT FROM authors_cursor
END

CLOSE authors_cursor
DEALLOCATE authors_cursor
GO
```

```
au_lname
----------------------------------------
Bennet
au_lname
----------------------------------------
Blotchet-Halls
au_lname
----------------------------------------
```

B. Utilizar FETCH para almacenar valores en variables

Este ejemplo es similar al ejemplo anterior, excepto en que la salida de las instrucciones FETCH se almacena en variables locales en lugar de ser devueltas directamente al cliente. La instrucción PRINT combina las variables en una misma cadena y la devuelve al cliente.

```
USE pubs
GO

-- Declare the variables to store the values returned by FETCH.
DECLARE @au_lname varchar(40), @au_fname varchar(20)


DECLARE authors_cursor CURSOR FOR
SELECT au_lname, au_fname FROM authors
WHERE au_lname LIKE "B%"
ORDER BY au_lname, au_fname

OPEN authors_cursor

-- Perform the first fetch and store the values in variables.
-- Note: The variables are in the same order as the columns
-- in the SELECT statement.

FETCH NEXT FROM authors_cursor
INTO @au_lname, @au_fname

-- Check @@FETCH_STATUS to see if there are any more rows to fetch.
WHILE @@FETCH_STATUS = 0
BEGIN

   -- Concatenate and display the current values in the variables.
   PRINT "Author: " + @au_fname + " " +  @au_lname

   -- This is executed as long as the previous fetch succeeds.
   FETCH NEXT FROM authors_cursor
   INTO @au_lname, @au_fname
END

CLOSE authors_cursor
DEALLOCATE authors_cursor
GO

Author: Abraham Bennet
Author: Reginald Blotchet-Halls
```

C. Declarar un cursor SCROLL y utilizar el resto de las opciones de FETCH

Este ejemplo crea un cursor SCROLL para permitir todas las posibilidades de desplazamiento con las opciones LAST, PRIOR, RELATIVE y ABSOLUTE.

```
USE pubs
GO

-- Execute the SELECT statement alone to show the
-- full result set that is used by the cursor.
SELECT au_lname, au_fname FROM authors
ORDER BY au_lname, au_fname

-- Declare the cursor.
DECLARE authors_cursor SCROLL CURSOR FOR
SELECT au_lname, au_fname FROM authors
ORDER BY au_lname, au_fname

OPEN authors_cursor

-- Fetch the last row in the cursor.
FETCH LAST FROM authors_cursor

-- Fetch the row immediately prior to the current row in the cursor.
FETCH PRIOR FROM authors_cursor

-- Fetch the second row in the cursor.
FETCH ABSOLUTE 2 FROM authors_cursor

-- Fetch the row that is three rows after the current row.
FETCH RELATIVE 3 FROM authors_cursor

-- Fetch the row that is two rows prior to the current row.
FETCH RELATIVE -2 FROM authors_cursor

CLOSE authors_cursor
DEALLOCATE authors_cursor
GO
```

```
au_lname                                 au_fname
---------------------------------------- --------------------
Bennet                                   Abraham
Blotchet-Halls                           Reginald
Carson                                   Cheryl
DeFrance                                 Michel
del Castillo                             Innes
Dull                                     Ann
Green                                    Marjorie
Greene                                   Morningstar
Gringlesby                               Burt
Hunter                                   Sheryl
Karsen                                   Livia
Locksley                                 Charlene
MacFeather                               Stearns
McBadden                                 Heather
O'Leary                                  Michael
```

```
Panteley                                  Sylvia
Ringer                                    Albert
Ringer                                    Anne
Smith                                     Meander
Straight                                  Dean
Stringer                                  Dirk
White                                     Johnson
Yokomoto                                  Akiko

au_lname                                  au_fname
---------------------------------------- --------------------
Yokomoto                                  Akiko
au_lname                                  au_fname
---------------------------------------- --------------------
White                                     Johnson
au_lname                                  au_fname
---------------------------------------- --------------------
Blotchet-Halls                            Reginald
au_lname                                  au_fname
---------------------------------------- --------------------
del Castillo                              Innes
au_lname                                  au_fname
---------------------------------------- --------------------
Carson                                    Cheryl
```

DECLARE CURSOR

Define los atributos de un cursor de servidor Transact-SQL, como su comportamiento de desplazamiento y la consulta utilizada para generar el conjunto de resultados sobre el que opera el cursor. DECLARE CURSOR acepta la sintaxis basada en el estándar SQL-92 y la sintaxis de un conjunto de extensiones de Transact-SQL.

Sintaxis extendida de Transact-SQL

DECLARE *cursor_name* CURSOR
[LOCAL | GLOBAL]
[FORWARD_ONLY | SCROLL]
[STATIC | KEYSET | DYNAMIC | FAST_FORWARD]
[READ_ONLY | SCROLL_LOCKS | OPTIMISTIC]
[TYPE_WARNING]
FOR *select_statement*
[FOR UPDATE [OF *column_name* [**,**...*n*]]]

Argumentos extendidos de Transact-SQL

cursor_name

Se trata del nombre del cursor de servidor Transact-SQL que se va a definir. El argumento *cursor_name* debe seguir las reglas de los identificadores.

LOCAL

Especifica que el alcance del cursor es local para el proceso por lotes, procedimiento almacenado o desencadenador en que se creó el cursor. El nombre del cursor sólo es válido dentro de este alcance. Es posible hacer referencia al cursor mediante variables de cursor locales del proceso por lotes, procedimiento almacenado, desencadenador o parámetro OUTPUT del procedimiento almacenado. El parámetro OUTPUT se utiliza para devolver el cursor local al proceso por lotes, procedimiento almacenado o desencadenador que realiza la llamada, el cual puede asignar el parámetro a una variable de cursor para hacer referencia al cursor después de finalizar el procedimiento almacenado. La asignación del cursor se cancela implícitamente cuando el proceso por lotes, procedimiento almacenado o desencadenador finalizan, a menos que el cursor se haya devuelto en un parámetro OUTPUT. En ese caso, se cancela la asignación del cursor cuando se cancela la asignación de la última variable que le hace referencia o ésta se sale del alcance.

GLOBAL

Especifica que el alcance del cursor es global para la conexión. Puede hacerse referencia al nombre del cursor en cualquier procedimiento almacenado o proceso por lotes que se ejecute durante la conexión. Sólo se cancela implícitamente la asignación del cursor cuando se realiza la desconexión.

> **Nota** Si no se especifica GLOBAL o LOCAL, el valor predeterminado se controla mediante la configuración de la opción de base de datos **cursor local como predeterminado**. En SQL Server versión 7.0, el valor predeterminado de esta opción es FALSE (falso) para que coincida con las versiones anteriores de SQL Server, donde todos los cursores eran globales. El valor predeterminado de esta opción puede cambiar en futuras versiones de SQL Server.

FORWARD_ONLY

Especifica que el cursor sólo se puede desplazar desde la primera a la última fila. FETCH NEXT es la única opción de recuperación aceptada. Si se especifica FORWARD_ONLY sin las palabras clave STATIC, KEYSET o DYNAMIC, el cursor funciona como un cursor DYNAMIC. Cuando no se especifica FORWARD_ONLY ni tampoco SCROLL, FORWARD_ONLY es la opción predeterminada, salvo si se incluyen las palabras clave STATIC, KEYSET o DYNAMIC. Los cursores STATIC, KEYSET y DYNAMIC toman como valor predeterminado SCROLL. A diferencia de las API de bases de datos como ODBC y ADO, FORWARD_ONLY se puede utilizar con los cursores STATIC, KEYSET y DYNAMIC de Transact-SQL. FAST_FORWARD y FORWARD_ONLY se excluyen mutuamente: si se especifica uno, no se puede especificar el otro.

STATIC

Define un cursor que hace una copia temporal de los datos que utiliza. Todas las peticiones al cursor se responden desde esta tabla temporal de **tempdb**; por ello, las modificaciones realizadas en las tablas base no se reflejarán en los datos obtenidos en las recuperaciones realizadas en el cursor y además este cursor no admite modificaciones.

KEYSET

Especifica que la pertenencia y el orden de las filas del cursor se fijan al abrir éste. El conjunto de claves que identifica de forma única las filas está integrado en una tabla de **tempdb** conocida como **keyset**. Los cambios en valores que no sean claves de las tablas base, ya sean realizados por el propietario del cursor o confirmados por otros usuarios, son visibles cuando el propietario se desplaza por el cursor. Las inserciones realizadas por otros usuarios no son visibles (no es posible hacer inserciones a través de un cursor de servidor Transact-SQL). Si se elimina una fila, el intento de recuperarla obtendrá un valor de -2 en @@FETCH_STATUS. Las actualizaciones de los valores de claves desde fuera del cursor se asemejan a la eliminación de la fila antigua seguida de la inserción de la nueva. La fila con los nuevos valores no es visible y los intentos de recuperar la de los valores antiguos devuelven el valor -2 en @@FETCH_STATUS. Los nuevos valores son visibles si la actualización se realiza a través del cursor, al especificar la cláusula WHERE CURRENT OF.

DYNAMIC

Define un cursor que, al desplazarse por él, refleja en su conjunto de resultados todos los cambios realizados en los datos de las filas. Los valores de los datos, el orden y la pertenencia de las filas pueden cambiar en cada búsqueda. La opción de recuperación ABSOLUTE no se puede utilizar en los cursores dinámicos.

FAST_FORWARD

Especifica un cursor FORWARD_ONLY, READ_ONLY con las optimizaciones de rendimiento habilitadas. No se puede especificar FAST_FORWARD si se especifica también SCROLL o FOR_UPDATE. FAST_FORWARD y FORWARD_ONLY se excluyen mutuamente: si se especifica uno, no se puede especificar el otro.

READ_ONLY

Evita que se efectúen actualizaciones a través de este cursor. No es posible hacer referencia al cursor en una cláusula WHERE CURRENT OF de una instrucción UPDATE o DELETE. Esta opción suplanta la posibilidad predeterminada de actualizar el cursor.

SCROLL_LOCKS

Especifica que el éxito de las actualizaciones o eliminaciones con posición, realizadas a través del cursor, está garantizado. Microsoft® SQL Server™ bloquea las filas al leerlas en

el cursor, lo que asegura su disponibilidad para posteriores modificaciones. No es posible especificar SCROLL_LOCKS si se incluye también FAST_FORWARD.

OPTIMISTIC

Especifica que las actualizaciones o eliminaciones con posición realizadas a través del cursor no tendrán éxito si la fila se ha actualizado después de ser leída en el cursor. SQL Server no bloquea las filas al leerlas en el cursor. En su lugar, utiliza comparaciones de valores de columna **timestamp** o un valor de suma de comprobación si la tabla no tiene columnas **timestamp**, para determinar si la fila se ha modificado después de leerla en el cursor. Si la fila se ha modificado, la actualización o eliminación con posición fracasa. No es posible especificar OPTIMISTIC si se incluye también FAST_FORWARD.

TYPE_WARNING

Especifica que se envía un mensaje de advertencia al cliente si el cursor se convierte implícitamente del tipo solicitado a otro.

select_statement

Es una instrucción SELECT estándar que define el conjunto de resultados del cursor. Las palabras clave COMPUTE, COMPUTE BY, FOR BROWSE e INTO no están permitidas en el argumento *select_statement* de una declaración de cursor.

SQL Server convierte implícitamente el cursor a otro tipo si las cláusulas de *select_statement* entran en conflicto con la funcionalidad del tipo de cursor solicitado.

UPDATE [OF *column_name* [**,**...*n*]]

Define las columnas actualizables en el cursor. Si se especifica el argumento OF *column_name* [**,**...*n*], sólo se podrán modificar las columnas incluidas en la lista. Si se especifica el argumento UPDATE sin una lista de columnas, se pueden actualizar todas las columnas, a menos que se haya especificado la opción de simultaneidad READ_ONLY.

Observaciones

DECLARE CURSOR define los atributos de un cursor de servidor Transact-SQL, como su comportamiento de desplazamiento y la consulta utilizada para generar el conjunto de resultados en que opera el cursor. La instrucción OPEN llena el conjunto de resultados y la instrucción FETCH devuelve una fila del conjunto de resultados. La instrucción CLOSE libera el conjunto de resultados actual asociado con el cursor. La instrucción DEALLOCATE libera los recursos que utiliza el cursor.

La primera forma de la instrucción DECLARE CURSOR utiliza la sintaxis de SQL-92 para declarar comportamientos de cursores. La segunda forma de DECLARE CURSOR utiliza extensiones de Transact-SQL que permiten definir cursores con los mismos tipos de cursor

utilizados en las funciones de cursores de la API de bases de datos de ODBC, ADO y Bibliotecas de bases de datos.

No puede combinar las dos formas. Si especifica las palabras clave SCROLL o INSENSITIVE antes de la palabra clave CURSOR, no puede utilizar ninguna palabra clave entre CURSOR y FOR *select_statement*. Si especifica palabras clave entre CURSOR y FOR *select_statement*, no puede especificar SCROLL o INSENSITIVE antes de la palabra clave CURSOR.

Si la instrucción DECLARE CURSOR con sintaxis de Transact-SQL no especifica READ_ONLY, OPTIMISTIC o SCROLL_LOCKS, el valor predeterminado es el siguiente:

- Si la instrucción SELECT no acepta actualizaciones (permisos insuficientes, acceso a tablas remotas que no aceptan actualizaciones, etc.), el cursor es de tipo READ_ONLY.
- El valor predeterminado de los cursores de tipo STATIC y FAST_FORWARD es READ_ONLY.
- El valor predeterminado de los cursores de tipo KEYSET y DYNAMIC es OPTIMISTIC.

Sólo se puede hacer referencia a nombres de cursores mediante otras instrucciones Transact-SQL. No se puede hacer referencia a los nombres de cursores mediante funciones de la API de base de datos. Por ejemplo, después de declarar un cursor, no se puede hacer referencia al nombre del cursor desde funciones o métodos de OLE DB, ODBC, ADO o Bibliotecas de bases de datos. No se pueden recuperar las filas del cursor con las funciones o métodos de recuperación de las API; las filas sólo se pueden recuperar mediante instrucciones FETCH de Transact-SQL.

Una vez que se ha declarado un cursor, se pueden utilizar estos procedimientos almacenados del sistema para determinar las características del cursor.

Procedimiento almacenado del sistema	**Descripción**
sp_cursor_list	Devuelve la lista de cursores visibles actualmente en la conexión y sus atributos.
sp_describe_cursor	Describe los atributos de un cursor, por ejemplo si es de desplazamiento sólo hacia delante o de desplazamiento.
sp_describe_cursor_columns	Describe los atributos de las columnas en el conjunto de resultados del cursor.
sp_describe_cursor_tables	Describe las tablas base a las que tiene acceso el

	cursor.

Se pueden utilizar variables como parte de la instrucción *select_statement* que declara un cursor. Sin embargo, los cambios a estas variables después de haber declarado el cursor no afectarán a la operación del cursor.

Ejemplos

A. Utilizar cursores simples y su sintaxis

El conjunto de resultados generado al abrir este cursor contiene todas las filas y todas las columnas de la tabla **authors** de la base de datos **pubs**. Este cursor se puede actualizar, y todas las actualizaciones y eliminaciones se representan en las recuperaciones realizadas contra el cursor. FETCH NEXT es la única recuperación disponible debido a que no se ha especificado la opción SCROLL.

```
DECLARE authors_cursor CURSOR
   FOR SELECT * FROM authors
OPEN authors_cursor
FETCH NEXT FROM authors_cursor
```

B. Utilizar cursores anidados para elaborar resultados de informes

Este ejemplo muestra cómo se pueden anidar los cursores para elaborar informes complejos. El cursor interno se declara para cada autor.

```
SET NOCOUNT ON

DECLARE @au_id varchar(11), @au_fname varchar(20), @au_lname varchar(40),
   @message varchar(80), @title varchar(80)

PRINT "-------- Utah Authors report --------"

DECLARE authors_cursor CURSOR FOR
SELECT au_id, au_fname, au_lname
FROM authors
WHERE state = "UT"
ORDER BY au_id

OPEN authors_cursor

FETCH NEXT FROM authors_cursor
INTO @au_id, @au_fname, @au_lname

WHILE @@FETCH_STATUS = 0
BEGIN
   PRINT " "
   SELECT @message = "----- Books by Author: " +
      @au_fname + " " + @au_lname
```

```
   PRINT @message

   -- Declare an inner cursor based
   -- on au_id from the outer cursor.

   DECLARE titles_cursor CURSOR FOR
   SELECT t.title
   FROM titleauthor ta, titles t
   WHERE ta.title_id = t.title_id AND
   ta.au_id = @au_id   -- Variable value from the outer cursor

   OPEN titles_cursor
   FETCH NEXT FROM titles_cursor INTO @title

   IF @@FETCH_STATUS <> 0
      PRINT "         <<No Books>>"

   WHILE @@FETCH_STATUS = 0
   BEGIN

      SELECT @message = "         " + @title
      PRINT @message
      FETCH NEXT FROM titles_cursor INTO @title

   END

   CLOSE titles_cursor
   DEALLOCATE titles_cursor

   -- Get the next author.
   FETCH NEXT FROM authors_cursor
   INTO @au_id, @au_fname, @au_lname
END

CLOSE authors_cursor
DEALLOCATE authors_cursor
GO

-------- Utah Authors report --------

----- Books by Author: Anne Ringer
         The Gourmet Microwave
         Is Anger the Enemy?

----- Books by Author: Albert Ringer
         Is Anger the Enemy?
         Life Without Fear
```

OPEN

Abre un cursor del servidor Transact-SQL y lo llena ejecutando la instrucción Transact-SQL especificada en la instrucción DECLARE CURSOR o SET *cursor_variable*.

Sintaxis

OPEN { { [GLOBAL] *cursor_name* } | *cursor_variable_name* }

Argumentos

GLOBAL

Especifica que *cursor_name* hace referencia a un cursor global.

cursor_name

Es el nombre de un cursor declarado. Si existen un cursor global y otro local con nombre *cursor_name*, éste hace referencia al cursor global si se especifica GLOBAL; en caso contrario, *cursor_name* hace referencia al cursor local.

cursor_variable_name

Es el nombre de la variable cursor que hace referencia a un cursor.

Observaciones

Si se declara el cursor con la opción INSENSITIVE o STATIC, OPEN crea una tabla temporal para mantener el conjunto de resultados. OPEN produce un error si el tamaño de cualquier fila en el conjunto de resultados excede el tamaño máximo de fila para las tablas de Microsoft® SQL Server™. Si el cursor se declara con la opción KEYSET, OPEN crea una tabla temporal para mantener el conjunto de claves. Las tablas temporales se almacenan en **tempdb**.

Después de abrir un cursor, utilice la función @@CURSOR_ROWS para recuperar el número de filas habilitadas en el último cursor abierto. Dependiendo del número de filas esperadas en el conjunto de resultados, SQL Server puede elegir llenar asincrónicamente un cursor controlado por conjuntos de claves en un subproceso separado. Esto permite a las búsquedas continuar inmediatamente, incluso si el conjunto de claves no está completamente lleno.

Para establecer los umbrales en los que SQL Server genera asincrónicamente los conjunto de claves, establezca la opción de configuración **cursor threshold**.

Ejemplos

Este ejemplo abre un cursor y busca todas sus filas.

```
DECLARE Employee_Cursor CURSOR FOR
SELECT LastName, FirstName
FROM Northwind.dbo.Employees
WHERE LastName like 'B%'

OPEN Employee_Cursor
```

```
FETCH NEXT FROM Employee_Cursor
WHILE @@FETCH_STATUS = 0
BEGIN
    FETCH NEXT FROM Employee_Cursor
END

CLOSE Employee_Cursor
DEALLOCATE Employee_Cursor
```

CLOSE

Cierra un cursor abierto liberando el conjunto actual de resultados y todos los bloqueos mantenidos sobre las filas en las que está colocado el cursor. CLOSE deja las estructuras de datos accesibles para que se puedan volver a abrir, pero las recuperaciones y las actualizaciones con posición no se permiten hasta que se vuelva a abrir el cursor. CLOSE se tiene que ejecutar sobre un cursor abierto; no se permite sobre cursores que sólo están declarados o que están cerrados.

Sintaxis

CLOSE { { [GLOBAL] *cursor_name* } | *cursor_variable_name* }

Argumentos

GLOBAL

Especifica que *cursor_name* hace referencia a un cursor global.

cursor_name

Es el nombre de un cursor abierto. Si *cursor_name* es el nombre de un cursor global y de un cursor local, *cursor_name* hace referencia al cursor global si se especifica GLOBAL; en caso contrario, *cursor_name* hace referencia al cursor local.

cursor_variable_name

Es el nombre de una variable de cursor asociada con un cursor abierto.

Ejemplos

Este ejemplo muestra la posición correcta de la instrucción CLOSE en un proceso de cursores.

```
USE pubs
GO

DECLARE authorcursor CURSOR FOR
SELECT au_fname, au_lname
```

```
FROM authors
ORDER BY au_fname, au_lname

OPEN authorcursor
FETCH NEXT FROM authorcursor
WHILE @@FETCH_STATUS = 0
BEGIN
   FETCH NEXT FROM authorcursor
END

CLOSE authorcursor
DEALLOCATE authorcursor
GO
```

DEALLOCATE

Quita una referencia a un cursor. Cuando se ha quitado la última referencia al cursor, Microsoft® SQL Server™ libera las estructuras de datos que componen el cursor.

Sintaxis

DEALLOCATE { { [GLOBAL] *cursor_name* } | *@cursor_variable_name* }

Argumentos

cursor_name

Es el nombre de un cursor ya declarado. Si hay un cursor global y otro local con el nombre *cursor_name*, este argumento hace referencia al cursor global si se especifica GLOBAL; de lo contrario, *cursor_name* hace referencia al cursor local.

@cursor_variable_name

Es el nombre de una variable de **cursor**. *@cursor_variable_name* debe ser del tipo **cursor**.

Observaciones

Las instrucciones que realizan operaciones sobre cursores utilizan un nombre de cursor o una variable de cursor para hacer referencia al cursor. DEALLOCATE quita la asociación existente entre un cursor y el nombre del cursor o la variable de cursor. Si un nombre o variable es el último que hace referencia a un cursor, se quita el cursor y se liberan los recursos que utiliza. Los bloqueos de desplazamiento utilizados para proteger el aislamiento de las recuperaciones se liberan en DEALLOCATE. Los bloqueos de transacciones utilizados para proteger las actualizaciones, incluidas las actualizaciones por posición creadas a través del cursor, se mantienen hasta el final de la transacción.

La instrucción DECLARE CURSOR asigna y asocia un cursor con un nombre de cursor:

DECLARE abc SCROLL CURSOR FOR
SELECT * FROM authors

Después de asociar un nombre de cursor con un cursor, ningún otro cursor del mismo alcance (GLOBAL o LOCAL) puede utilizar el nombre hasta que se haya cancelado la asignación al cursor.

Una variable de cursor se puede asociar con un cursor mediante dos métodos:

- Por nombre con una instrucción SET que asocia un cursor con una variable de cursor:
- DECLARE @MyCrsrRef CURSOR
- SET @MyCrsrRef = abc
- También se puede crear y asociar un cursor con una variable sin necesidad de definir un nombre de cursor:
- DECLARE @MyCursor CURSOR
- SET @MyCursor = CURSOR LOCAL SCROLL FOR
- SELECT * FROM titles

La instrucción DEALLOCATE*@cursor_variable_name* quita sólo la referencia de la variable mencionada al cursor. No se cancela la asignación de la variable hasta que sale de alcance al final del proceso por lotes, procedimiento almacenado o desencadenador. Después de una instrucción DEALLOCATE *@cursor_variable_name*, se puede asociar la variable con otro cursor mediante la instrucción SET.

```
USE pubs
GO
DECLARE @MyCursor CURSOR
SET @MyCursor = CURSOR LOCAL SCROLL FOR
SELECT * FROM titles

DEALLOCATE @MyCursor

SET @MyCursor = CURSOR LOCAL SCROLL FOR
SELECT * FROM sales
GO
```

No es necesario cancelar explícitamente la asignación de una variable de cursor. La asignación de la variable se cancela implícitamente cuando sale de alcance.

Ejemplos

Esta secuencia de comandos muestra cómo los cursores se mantienen hasta que se ha cancelado la asignación del último nombre o variable que hace referencia a ellos.

```
USE pubs
GO
-- Create and open a global named cursor that
-- is visible outside the batch.
```

```
DECLARE abc CURSOR GLOBAL SCROLL FOR
SELECT * FROM authors
OPEN abc
GO
-- Reference the named cursor with a cursor variable.
DECLARE @MyCrsrRef1 CURSOR
SET @MyCrsrRef1 = abc
-- Now deallocate the cursor reference.
DEALLOCATE @MyCrsrRef1
-- Cursor abc still exists.
FETCH NEXT FROM abc
GO
-- Reference the named cursor again.
DECLARE @MyCrsrRef2 CURSOR
SET @MyCrsrRef2 = abc
-- Now deallocate cursor name abc.
DEALLOCATE abc
-- Cursor still exists, referenced by @MyCrsrRef2.
FETCH NEXT FROM @MyCrsrRef2
-- Cursor finally is deallocated when last referencing
-- variable goes out of scope at the end of the batch.
GO
-- Create an unnamed cursor.
DECLARE @MyCursor CURSOR
SET @MyCursor = CURSOR LOCAL SCROLL FOR
SELECT * FROM titles
-- The following statement deallocates the cursor
-- because no other variables reference it.
DEALLOCATE @MyCursor
GO
```

TRANSACCIONALIDAD

BEGIN TRANSACTION

Marca el punto de inicio de una transacción local explícita. La instrucción BEGIN TRANSACTION incrementa @@TRANCOUNT en 1.

Sintaxis

BEGIN TRAN [SACTION] [*transaction_name* | *@tran_name_variable*
[WITH MARK ['*description*']]]

Argumentos

transaction_name

Es el nombre asignado a la transacción. El argumento *transaction_name* debe cumplir las reglas de los identificadores, pero los identificadores de más de 32 caracteres no se admiten. Utilice nombres de transacciones solamente en la pareja más externa de instrucciones BEGIN...COMMIT o BEGIN...ROLLBACK anidadas.

@*tran_name_variable*

Se trata del nombre de una variable definida por el usuario que contiene un nombre de transacción válido. La variable se debe declarar con un tipo de datos **char**, **varchar**, **nchar** o **nvarchar**.

WITH MARK ['*description*']

Especifica que la transacción está marcada en el registro. *description* es una cadena que describe la marca.

Si utiliza WITH MARK, debe especificar un nombre de transacción. WITH MARK permite restaurar un registro de transacciones hasta una marca con nombre.

Observaciones

BEGIN TRANSACTION representa un punto en el que los datos a los que hace referencia una conexión son lógica y físicamente coherentes. Si se producen errores, se pueden deshacer todas las modificaciones realizadas en los datos después de BEGIN TRANSACTION para devolver los datos al estado conocido de coherencia. Cada transacción dura hasta que se completa sin errores y se emite COMMIT TRANSACTION para hacer que las modificaciones sean una parte permanente de la base de datos, o hasta que se producen errores y se borran todas las modificaciones con la instrucción ROLLBACK TRANSACTION.

BEGIN TRANSACTION inicia una transacción local para la conexión que emite la instrucción. Según la configuración del nivel de aislamiento de la transacción actual, la transacción bloquea muchos recursos adquiridos para aceptar las instrucciones Transact-SQL que emite la conexión hasta que finaliza con una instrucción COMMIT TRANSACTION o ROLLBACK TRANSACTION. Las transacciones que quedan pendientes durante mucho tiempo pueden impedir que otros usuarios tengan acceso a estos recursos compartidos.

Aunque BEGIN TRANSACTION inicia una transacción local, ésta no se guardará en el registro de transacciones hasta que la aplicación realice posteriormente una acción que se deba grabar en el registro, como la ejecución de una instrucción INSERT, UPDATE o DELETE. Una aplicación puede realizar acciones como, por ejemplo, adquirir bloqueos para proteger el nivel de aislamiento de transacciones de instrucciones SELECT, pero no se guarda ningún dato en el registro hasta que la aplicación realiza una acción.

Asignar un nombre a varias transacciones en un conjunto de transacciones anidadas tiene poco efecto sobre la transacción. Solamente el nombre de la primera transacción (la más externa) se registra en el sistema. Deshacer a otro nombre (distinto a un nombre de punto guardado válido) genera un error. De hecho, no se deshace ninguna de las instrucciones ejecutadas antes de la operación de deshacer en el momento en que se produce este error. Sólo se deshacen las instrucciones cuando se deshace la transacción externa.

BEGIN TRANSACTION inicia una transacción local. La transacción local aumenta al nivel de transacción distribuida si se realizan las siguientes acciones antes de confirmarla o deshacerla:

- Se ejecuta una instrucción INSERT, DELETE o UPDATE que hace referencia a una tabla remota de un servidor vinculado. La instrucción INSERT, UDPATE o DELETE causa un error si el proveedor de OLE DB utilizado para obtener acceso al servidor vinculado no es compatible con la interfaz **ITransactionJoin**.

- Se realiza una llamada a un procedimiento almacenado remoto cuando la opción REMOTE_PROC_TRANSACTIONS es ON.

La copia local de SQL Server se convierte en el controlador de la transacción y utiliza MS DTC para administrar la transacción distribuida.

Transacciones marcadas

La opción WITH MARK coloca el nombre de la transacción en el registro de transacciones. Al restaurar una base de datos a su estado anterior, se puede utilizar la transacción marcada en lugar de la fecha y la hora.

Además, se necesitan las marcas del registro de transacciones si tiene la intención de recuperar un conjunto de bases de datos relacionadas a un estado consistente lógicamente. Una transacción distribuida puede colocar las marcas en los registros de transacción de las bases de datos relacionadas. Recuperar el conjunto de bases de datos relacionales hasta

estas marcas da como resultado un conjunto de bases de datos consistente en cuanto a las transacciones. La colocación de las marcas en las bases de datos relacionadas requiere procedimientos especiales. La marca se coloca en el registro de transacciones solamente si la transacción marcada actualiza la base de datos. No se marcan las transacciones que no modifican los datos.

BEGIN TRAN *new_name* WITH MARK puede anidarse en cualquier transacción existente no marcada. De ese modo, *new_name* se convierte en el nombre de marca de la transacción, aunque esa transacción ya tenga uno. En el siguiente ejemplo, M2 es el nombre de la marca.

```
BEGIN TRAN T1
UPDATE table1 ...
BEGIN TRAN M2 WITH MARK
UPDATE table2 ...
SELECT * from table1
COMMIT TRAN M2
UPDATE table3 ...
COMMIT TRAN T1
```

Al intentar marcar una transacción ya marcada se produce un mensaje de advertencia (no de error):

```
BEGIN TRAN T1 WITH MARK
UPDATE table1 ...
BEGIN TRAN M2 WITH MARK

Server: Msg 3920, Level 16, State 1, Line 3
WITH MARK option only applies to the first BEGIN TRAN WITH MARK.
The option is ignored.
```

Ejemplos

A. Asignar un nombre a una transacción

Este ejemplo muestra cómo asignar un nombre a una transacción. Al confirmar la transacción con nombre, aumentan un 10 por ciento los beneficios pagados por todos los libros de informática más vendidos.

```
DECLARE @TranName VARCHAR(20)
SELECT @TranName = 'MyTransaction'

BEGIN TRANSACTION @TranName
GO
USE pubs
GO
UPDATE roysched
SET royalty = royalty * 1.10
WHERE title_id LIKE 'Pc%'
GO

COMMIT TRANSACTION MyTransaction
```

```
GO
```

B. Marcar una transacción

Este ejemplo muestra cómo marcar una transacción. Se marca la transacción llamada "RoyaltyUpdate".

```
BEGIN TRANSACTION RoyaltyUpdate
   WITH MARK 'Update royalty values'
GO
USE pubs
GO
UPDATE roysched
   SET royalty = royalty * 1.10
   WHERE title_id LIKE 'Pc%'
GO
COMMIT TRANSACTION RoyaltyUpdate
GO
```

COMMIT TRANSACTION

Marca el final de una transacción correcta, implícita o definida por el usuario. Si @@TRANCOUNT es 1, COMMIT TRANSACTION hace que todas las modificaciones efectuadas sobre los datos desde el inicio de la transacción sean parte permanente de la base de datos, libera los recursos mantenidos por la conexión y reduce @@TRANCOUNT a 0. Si @@TRANCOUNT es mayor que 1, COMMIT TRANSACTION sólo reduce @@TRANCOUNT en 1.

Sintaxis

COMMIT [TRAN [SACTION] [*transaction_name* | **@***tran_name_variable*]]

Argumentos

transaction_name

Microsoft® SQL Server™ lo omite. *transaction_name* especifica un nombre de transacción asignado por una previa instrucción BEGIN TRANSACTION. *transaction_name* tiene que cumplir las reglas de definición de identificadores, pero sólo se utilizan sus 32 primeros caracteres. *transaction_name* se puede utilizar como ayuda al programador, indicándole con qué instrucción BEGIN TRANSACTION anidada está asociada la instrucción COMMIT TRANSACTION.

@tran_name_variable

Se trata del nombre de una variable definida por el usuario que contiene un nombre de transacción válido. La variable se debe declarar con un tipo de datos **char**, **varchar**, **nchar** o **nvarchar**.

Observaciones

Es responsabilidad del programador de Transact-SQL utilizar COMMIT TRANSACTION sólo en el punto donde todos los datos a los que hace referencia la transacción sean lógicamente correctos.

Si la transacción que se ha confirmado era una transacción Transact-SQL distribuida, COMMIT TRANSACTION hace que MS DTC utilice el protocolo de confirmación en dos fases para enviar confirmaciones a los servidores involucrados en la transacción. Si una transacción local afecta a dos o más bases de datos del mismo servidor, SQL Server utiliza una confirmación interna en dos fases para confirmar todas las bases de datos involucradas en la transacción.

Cuando se utiliza en transacciones anidadas, las confirmaciones de las transacciones anidadas no liberan recursos ni hacen permanentes sus modificaciones. Las modificaciones sobre los datos sólo quedan permanentes y se liberan los recursos cuando se confirma la transacción más externa. Cada COMMIT TRANSACTION que se ejecute cuando @@TRANCOUNT sea mayor que 1 sólo reduce @@TRANCOUNT en 1. Cuando @@TRANCOUNT llega a 0, se confirma la transacción externa entera. Como SQL Server omite *transaction_name*, la ejecución de una instrucción COMMIT TRANSACTION que haga referencia al nombre de una transacción externa cuando haya transacciones anidadas pendientes sólo reduce @@TRANCOUNT en 1.

La ejecución de COMMIT TRANSACTION cuando @@TRANCOUNT es 0 produce un error que indica que no hay ninguna instrucción BEGIN TRANSACTION asociada.

No se puede cancelar una transacción después de ejecutar una instrucción COMMIT TRANSACTION, porque las modificaciones sobre los datos ya son parte permanente de la base de datos.

Ejemplos

A. Confirmar una transacción.

Este ejemplo incrementa el anticipo que se paga al autor si las ventas anuales de un libro son superiores a $8.000.

```
BEGIN TRANSACTION
USE pubs
GO
UPDATE titles
SET advance = advance * 1.25
WHERE ytd_sales > 8000
GO
```

```
COMMIT
GO
```

B. Confirmar una transacción anidada.

Este ejemplo crea una tabla, genera tres niveles de transacciones anidadas y después confirma la transacción anidada. Aunque la instrucción COMMIT TRANSACTION tiene el parámetro *transaction_name*, no hay relación entre las instrucciones COMMIT TRANSACTION y BEGIN TRANSACTION. Los parámetros *transaction_name* sólo son ayudas para que el programador pueda asegurarse de que escribe el número apropiado de confirmaciones para reducir @@TRANCOUNT hasta 0, confirmando así la transacción más externa.

```
CREATE TABLE TestTran (Cola INT PRIMARY KEY, Colb CHAR(3))
GO
BEGIN TRANSACTION OuterTran -- @@TRANCOUNT set to 1.
GO
INSERT INTO TestTran VALUES (1, 'aaa')
GO
BEGIN TRANSACTION Inner1 -- @@TRANCOUNT set to 2.
GO
INSERT INTO TestTran VALUES (2, 'bbb')
GO
BEGIN TRANSACTION Inner2 -- @@TRANCOUNT set to 3.
GO
INSERT INTO TestTran VALUES (3, 'ccc')
GO
COMMIT TRANSACTION Inner2 -- Decrements @@TRANCOUNT to 2.
-- Nothing committed.
GO
COMMIT TRANSACTION Inner1 -- Decrements @@TRANCOUNT to 1.
-- Nothing committed.
GO
COMMIT TRANSACTION OuterTran -- Decrements @@TRANCOUNT to 0.
-- Commits outer transaction OuterTran.
GO
```

ROLLBACK TRANSACTION

Deshace una transacción explícita o implícita hasta el inicio de la transacción o hasta un punto de almacenamiento dentro de una transacción.

Sintaxis

ROLLBACK [TRAN [SACTION]
[*transaction_name* | *@tran_name_variable*
| *savepoint_name* | *@savepoint_variable*]]

Argumentos

transaction_name

Es el nombre asignado a la transacción en BEGIN TRANSACTION. Se debe ajustar a las reglas para los identificadores, pero sólo se utilizan los primeros 32 caracteres del nombre de la transacción. Cuando se trata de transacciones anidadas, *transaction_name* debe ser el nombre de la instrucción BEGIN TRANSACTION más externa.

@tran_name_variable

Se trata del nombre de una variable definida por el usuario que contiene un nombre de transacción válido. La variable se debe declarar con un tipo de datos **char**, **varchar**, **nchar** o **nvarchar**.

savepoint_name

Es el punto de almacenamiento de una instrucción SAVE TRANSACTION y se debe ajustar a las reglas para los identificadores. Utilice *savepoint_name* cuando una operación condicional para deshacer sólo deba afectar a parte de la transacción.

@savepoint_variable

Es el nombre de una variable definida por el usuario que contiene un nombre de punto de almacenamiento válido. La variable se debe declarar con un tipo de datos **char**, **varchar**, **nchar** o **nvarchar**.

Observaciones

ROLLBACK TRANSACTION elimina todas las modificaciones de datos realizadas desde el inicio de la transacción o hasta un punto de almacenamiento. También libera los recursos que retiene la transacción.

ROLLBACK TRANSACTION sin un *savepoint_name* o *transaction_name* deshace todas las instrucciones hasta el principio de la transacción. Cuando se trata de transacciones anidadas, esta misma instrucción deshace todas las transacciones internas hasta la instrucción BEGIN TRANSACTION más externa. En ambos casos, ROLLBACK TRANSACTION disminuye la función del sistema @@TRANCOUNT a 0, mientras que ROLLBACK TRANSACTION con *savepoint_name* no disminuye @@TRANCOUNT.

Una instrucción ROLLBACK TRANSACTION que especifica un *savepoint_name* no libera ningún bloqueo.

ROLLBACK TRANSACTION no puede hacer referencia a un *savepoint_name* en transacciones distribuidas que se iniciaron explícitamente con BEGIN DISTRIBUTED TRANSACTION o que se escalaron a partir de una transacción local.

Una transacción no se puede deshacer después de ejecutar una instrucción COMMIT TRANSACTION.

En una transacción se permiten nombres de puntos de almacenamiento duplicados, pero una instrucción ROLLBACK TRANSACTION que utilice este tipo de nombre sólo deshace las transacciones realizadas hasta la instrucción SAVE TRANSACTION más reciente que también utilice este nombre.

En los procedimientos almacenados, las instrucciones ROLLBACK TRANSACTION sin un *savepoint_name* o *transaction_name* deshacen todas las instrucciones hasta la instrucción BEGIN TRANSACTION más externa. Una instrucción ROLLBACK TRANSACTION de un procedimiento almacenado produce un mensaje informativo, siempre que el procedimiento provoque que @@TRANCOUNT tenga, al finalizar el desencadenador, un valor diferente al que tenía cuando se llamó al procedimiento almacenado. Este mensaje no afecta a los siguientes procesos.

Si se emite la instrucción ROLLBACK TRANSACTION en un desencadenador:

- Se deshacen todas las modificaciones de datos realizadas hasta ese punto de la transacción actual, incluidas las que realizó el desencadenador.

- El desencadenador continúa la ejecución del resto de las instrucciones después de la instrucción ROLLBACK. Si alguna de estas instrucciones modifica datos, no se deshacen las modificaciones. La ejecución de las instrucciones restantes no activa ningún desencadenador anidado.

- Tampoco se ejecutan las instrucciones del lote después de la instrucción que activó el desencadenador.

@@TRANCOUNT se incrementa en uno al entrar en un desencadenador, incluso cuando está en modo de confirmación automática. (El sistema trata a un desencadenador como a una transacción anidada implícita.)

Las instrucciones ROLLBACK TRANSACTION de los procedimientos almacenados no afectan a las siguientes instrucciones del lote que llamó al procedimiento; se ejecutan las siguientes instrucciones del lote. Las instrucciones ROLLBACK TRANSACTION de los desencadenadores terminan el lote que contiene la instrucción que activó el desencadenador; no se ejecutan las siguientes instrucciones del lote.

Una instrucción ROLLBACK TRANSACTION no produce ningún mensaje para el usuario. Si necesita indicar advertencias en procedimientos almacenados o en desencadenadores, utilice las instrucciones RAISERROR o PRINT. RAISERROR es la instrucción más adecuada para indicar errores.

El efecto de ROLLBACK en los cursores se define mediante estas reglas:

1. Con CURSOR_CLOSE_ON_COMMIT establecido en ON, ROLLBACK cierra todos los cursores abiertos pero sin cancelar su asignación.

2. Con CURSOR_CLOSE_ON_COMMIT establecido en OFF, ROLLBACK no afecta a los cursores STATIC o INSENSITIVE sincrónicos abiertos o a los cursores STATIC asincrónicos que se hayan llenado completamente. Se cierran los cursores de otros tipos que estén abiertos, pero sin cancelar su asignación.

3. Un error que finaliza un lote y genera una operación de deshacer interna cancela la asignación de todos los cursores declarados en el lote que contiene la instrucción errónea. Se cancela la asignación de todos los cursores independientemente de su tipo o de la configuración de CURSOR_CLOSE_ON_COMMIT. Esto incluye a los cursores declarados en procedimientos almacenados a los que llama el lote con errores. Los cursores declarados en un lote antes del lote erróneo están sujetos a las reglas 1 y 2. Un error de interbloqueo constituye un ejemplo de este tipo de error. Una instrucción ROLLBACK emitida en un desencadenador también genera automáticamente este tipo de error.

SET TRANSACTION ISOLATION LEVEL

Controla el comportamiento de bloqueo predeterminado de todas las instrucciones SELECT de Microsoft® SQL Server™ ejecutadas en una conexión.

Sintaxis

```
SET TRANSACTION ISOLATION LEVEL
  { READ COMMITTED
    | READ UNCOMMITTED
    | REPEATABLE READ
    | SERIALIZABLE
  }
```

Argumentos

READ COMMITTED

Especifica que se mantengan los bloqueos compartidos mientras se leen datos para evitar lecturas no actualizadas, pero se pueden modificar los datos antes del final de la transacción, lo que provoca lecturas no repetibles o datos fantasma. Esta opción es la predeterminada en SQL Server.

READ UNCOMMITTED

Implementa las lecturas no confirmadas o el bloqueo de nivel de aislamiento 0, lo que significa que no hay bloqueos compartidos y que los bloqueos exclusivos no están garantizados. Cuando se establece esta opción, es posible leer datos no confirmados, los valores pueden cambiar y pueden aparecer y desaparecer filas en el conjunto de datos antes del final de la transacción. Esta opción tiene el mismo efecto que establecer NOLOCK en todas las tablas y en todas las instrucciones SELECT de la transacción. Se trata del menos restrictivo de los cuatro niveles de aislamiento.

REPEATABLE READ

Se establecen bloqueos para todos los datos utilizados en la consulta, lo que impide que otros usuarios los actualicen, aunque es posible insertar nuevas filas fantasmas en los datos que otro usuario establezca, de modo que se incluyan en lecturas posteriores de la misma transacción. Como la simultaneidad es inferior que el nivel de aislamiento predeterminado, sólo se debe usar esta opción cuando sea necesario.

SERIALIZABLE

Se establece un bloqueo de intervalo en el conjunto de datos, lo que impide que otros usuarios actualicen o inserten filas en el conjunto de datos hasta que finalice la transacción. Es el más restrictivo de los cuatro niveles de aislamiento. Al ser menor la simultaneidad, sólo se debe utilizar esta opción cuando sea necesario. Esta opción tiene el mismo efecto que establecer HOLDLOCK en todas las tablas y en todas las instrucciones SELECT de la transacción.

Observaciones

Sólo es posible establecer una de las opciones cada vez y permanecerá activa para la conexión hasta que se cambie explícitamente. La opción determina el comportamiento predeterminado, a menos que se especifique una opción de optimización en el nivel de tabla en la cláusula FROM de la instrucción.

La opción SET TRANSACTION ISOLATION LEVEL se establece en tiempo de ejecución, no en tiempo de análisis.

Ejemplos

En este ejemplo se establece TRANSACTION ISOLATION LEVEL para la sesión. En cada instrucción siguiente de Transact-SQL, SQL Server mantendrá todos los bloqueos compartidos hasta el final de la transacción.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO
BEGIN TRANSACTION
SELECT * FROM publishers
SELECT * FROM authors
...
COMMIT TRANSACTION
```

CONTROLES DE FLUJO

BEGIN...END

Encierra un conjunto de instrucciones Transact-SQL de forma que se pueda ejecutar. BEGIN y END son palabras clave del lenguaje de control de flujo.

Sintaxis

```
BEGIN
   {
      sql_statement
      | statement_block
   }
END
```

Argumentos

{*sql_statement* | *statement_block*}

Es cualquier instrucción o grupo de instrucciones Transact-SQL válidos definidos como bloque de instrucciones.

Observaciones

Los bloques BEGIN...END pueden anidarse.

Aunque todas las instrucciones Transact-SQL son válidas en un bloque BEGIN...END, ciertas instrucciones Transact-SQL no deben agruparse en el mismo proceso por lotes (bloque de instrucciones).

Ejemplos

En este ejemplo, BEGIN y END definen un conjunto de instrucciones Transact-SQL que se ejecutan juntas. Si no se incluye el bloque BEGIN...END, la condición IF causará que sólo se ejecute la instrucción ROLLBACK TRANSACTION y no se devolverá el mensaje impreso.

```
USE pubs
GO
CREATE TRIGGER deltitle
ON titles
FOR delete
AS
IF    (SELECT COUNT(*) FROM deleted, sales
      WHERE sales.title_id = deleted.title_id) > 0
   BEGIN
```

```
      ROLLBACK TRANSACTION
      PRINT 'You can't delete a title with sales.'
END
```

BREAK

Sale del bucle WHILE más interno. Las instrucciones que siguen a la palabra clave END se omiten. A menudo, pero no siempre, BREAK se activa mediante una prueba IF.

CONTINUE

Reinicia un bucle WHILE. Las instrucciones que se encuentren a continuación de la palabra clave CONTINUE se omiten. CONTINUE se suele activar, aunque no siempre, con una comprobación IF.

GOTO

Altera el flujo de ejecución y lo dirige a una etiqueta. Las instrucciones Transact-SQL que siguen a una instrucción GOTO se pasan por alto y el procesamiento continúa en el punto que marca la etiqueta. Las instrucciones GOTO y las etiquetas se pueden utilizar en cualquier punto de un procedimiento, lote o bloque de instrucciones. Las instrucciones GOTO se pueden anidar.

Sintaxis

Definición de la etiqueta:

label **:**

Alteración de la ejecución:

GOTO *label*

Argumentos

label

Es el punto a continuación del cual comienza el procesamiento cuando una instrucción GOTO especifica esa etiqueta. Las etiquetas se deben ajustar a las normas de los

identificadores. Las etiquetas se pueden utilizar como comentarios, tanto si se usa GOTO como si no.

Observaciones

GOTO puede aparecer dentro de las instrucciones de control de flujo condicional, en bloques de instrucciones o en procedimientos, pero no se puede dirigir a una etiqueta externa al lote. La ramificación con GOTO se puede dirigir a una etiqueta definida antes o después de la instrucción GOTO.

Ejemplos

Este ejemplo muestra un bucle con GOTO como alternativa al uso de WHILE.

> **Nota** El cursor **tnames_cursor** no está definido. Este ejemplo sólo tiene propósitos ilustrativos.

```
USE pubs
GO
DECLARE @tablename sysname
SET @tablename = N'authors'
table_loop:
   IF (@@FETCH_STATUS <> -2)
   BEGIN
      SELECT @tablename = RTRIM(UPPER(@tablename))
      EXEC ("SELECT """ + @tablename + """ = COUNT(*) FROM "
            + @tablename )
      PRINT " "
   END
   FETCH NEXT FROM tnames_cursor INTO @tablename
IF (@@FETCH_STATUS <> -1) GOTO table_loop
GO
```

IF...ELSE

Impone condiciones en la ejecución de una instrucción Transact-SQL. La instrucción Transact-SQL que sigue a una palabra clave IF y a su condición se ejecuta si la condición se satisface (cuando la expresión booleana devuelve TRUE). La palabra clave opcional ELSE introduce una instrucción Transact-SQL alternativa que se ejecuta cuando la condición IF no se satisface (cuando la expresión booleana devuelve FALSE).

Sintaxis

IF *Boolean_expression*
   { *sql_statement* | *statement_block* }

[ELSE
  { *sql_statement* | *statement_block* }]

Argumentos

Boolean_expression

Es una expresión que devuelve TRUE o FALSE. Si la expresión booleana contiene una instrucción SELECT, la instrucción SELECT debe ir entre paréntesis.

{*sql_statement* | *statement_block*}

Se trata de cualquier instrucción o grupo de instrucciones Transact-SQL definidos con un bloque de instrucciones. A menos que se utilice un bloque de instrucciones, la condición IF o ELSE puede afectar al rendimiento de una sola instrucción Transact-SQL. Para definir un bloque de instrucciones, utilice las palabras clave de control de flujo BEGIN y END. Las instrucciones CREATE TABLE o SELECT INTO deben hacer referencia al mismo nombre de tabla si se utilizan las instrucciones CREATE TABLE o SELECT INTO en las áreas IF y ELSE del bloque IF...ELSE.

Observaciones

Se pueden utilizar construcciones IF...ELSE en lotes, en procedimientos almacenados (en los que se utilizan a menudo estas construcciones para probar la existencia de algún parámetro) y en consultas *ad hoc*.

Las pruebas IF pueden estar anidadas después de otro IF o a continuación de un ELSE. No hay límite en el número de niveles anidados.

Ejemplos

A. Utilizar un bloque IF...ELSE

En este ejemplo se muestra una condición IF con un bloque de instrucciones. Si el precio promedio del título no es menor de 15 $, se imprime el texto: "Average title price is more than $15" (el precio promedio del título es mayor que 15 $).

```
USE pubs

IF (SELECT AVG(price) FROM titles WHERE type = 'mod_cook') < $15
BEGIN
   PRINT 'The following titles are excellent mod_cook books:'
   PRINT ' '
   SELECT SUBSTRING(title, 1, 35) AS Title
   FROM titles
   WHERE type = 'mod_cook'
END
ELSE
   PRINT 'Average title price is more than $15.'
```

El siguiente es el conjunto de resultados:

```
The following titles are excellent mod_cook books:

Title
-----------------------------------
Silicon Valley Gastronomic Treats
The Gourmet Microwave

(2 row(s) affected)
```

B. Utilizar más de un bloque IF...ELSE

En este ejemplo se utilizan dos bloques IF. Si el precio promedio del título no es menor de 15 $, se imprime el texto: "El precio promedio del título es mayor que 15 $". Si el precio promedio de los libros de cocina moderna es mayor que 15 $, se ejecuta la instrucción que imprime el mensaje que indica que los libros de cocina moderna son caros.

```
USE pubs

IF (SELECT AVG(price) FROM titles WHERE type = 'mod_cook') < $15
BEGIN
   PRINT 'The following titles are excellent mod_cook books:'
   PRINT ' '
   SELECT SUBSTRING(title, 1, 35) AS Title
   FROM titles
   WHERE type = 'mod_cook'
END
ELSE
   IF (SELECT AVG(price) FROM titles WHERE type = 'mod_cook') > $15
BEGIN
   PRINT 'The following titles are expensive mod_cook books:'
   PRINT ' '
   SELECT SUBSTRING(title, 1, 35) AS Title
   FROM titles
   WHERE type = 'mod_cook'
END
```

RETURN

Sale incondicionalmente de una consulta o procedimiento. RETURN es inmediata y completa, y se puede utilizar en cualquier punto para salir de un procedimiento, lote o bloque de instrucciones. Las instrucciones que siguen a RETURN no se ejecutan.

Sintaxis

RETURN [*integer_expression*]

Argumentos

integer_expression

Es el valor entero que se devuelve. Los procedimientos almacenados pueden devolver un valor entero al procedimiento que realiza la llamada o a una aplicación.

Tipos devueltos

Opcionalmente devuelve **int**.

> **Nota** A menos que se especifique lo contrario, todos los procedimientos almacenados del sistema devuelven el valor cero, que indica que fueron correctos; un valor distinto de cero indica que se ha producido un error.

Observaciones

Cuando se utiliza con un procedimiento almacenado, RETURN no puede devolver un valor NULL. Si un procedimiento intenta devolver un valor NULL (por ejemplo, al utilizar RETURN **@status** si **@status** es NULL), se genera un mensaje de advertencia y se devuelve un valor de 0.

El valor del estado de retorno se puede incluir en las siguientes instrucciones de Transact-SQL del lote o procedimiento que ejecutó el procedimiento actual, pero se deben introducir de la forma siguiente:

```
EXECUTE @return_status = procedure_name
```

Ejemplos

A. Retorno de un procedimiento

En este ejemplo se muestra que si no se proporciona el nombre del usuario como parámetro al ejecutar **findjobs**, RETURN provoca que el procedimiento salga tras enviar un mensaje a la pantalla del usuario. Si se proporciona un nombre de usuario, se obtienen de las tablas del sistema adecuadas los nombres de todos los objetos creados por este usuario en la base de datos actual.

```
CREATE PROCEDURE findjobs @nm sysname = NULL
AS
IF @nm IS NULL
   BEGIN
      PRINT 'You must give a username'
      RETURN
   END
ELSE
   BEGIN
      SELECT o.name, o.id, o.uid
      FROM sysobjects o INNER JOIN master..syslogins l
```

```
         ON o.uid = l.sid
      WHERE l.name = @nm
   END
```

B. Devolver códigos de estado

En este ejemplo se comprueba el estado del identificador del autor especificado. Si el estado es California (CA), se devuelve el estado 1. En caso contrario, se devuelve 2 para cualquier otra condición (un valor distinto de CA en **state** o un valor de **au_id** que no coincide con una fila).

```
CREATE PROCEDURE checkstate @param varchar(11)
AS
IF (SELECT state FROM authors WHERE au_id = @param) = 'CA'
   RETURN 1
ELSE
   RETURN 2
```

En los ejemplos siguientes se muestra el estado de retorno de la ejecución de **checkstate**. El primero muestra un autor de California; el segundo, un autor que no es de California y el tercero un autor no válido. Se debe declarar la variable local **@return_status** antes de poderla utilizar.

```
DECLARE @return_status int
EXEC @return_status = checkstate '172-32-1176'
SELECT 'Return Status' = @return_status
GO
```

El siguiente es el conjunto de resultados:

```
Return Status
-------------
1
```

Ejecute la consulta de nuevo con un número de autor diferente.

```
DECLARE @return_status int
EXEC @return_status = checkstate '648-92-1872'
SELECT 'Return Status' = @return_status
GO
```

El siguiente es el conjunto de resultados:

```
Return Status
-------------
2
```

Ejecute la consulta de nuevo con otro número de autor.

```
DECLARE @return_status int
EXEC @return_status = checkstate '12345678901'
```

```
SELECT 'Return Status' = @return_status
GO
```

El siguiente es el conjunto de resultados:

```
Return Status
-------------
2
```

WAITFOR

Especifica un tiempo, intervalo de tiempo o suceso que desencadena la ejecución de un bloque de instrucciones, procedimiento almacenado o transacción.

Sintaxis

WAITFOR { DELAY '*time*' | TIME '*time*' }

Argumentos

DELAY

Indica a Microsoft® SQL Server™ que espere hasta que haya transcurrido el tiempo especificado, hasta un máximo de 24 horas.

'*time*'

Es la cantidad de tiempo que se esperará. El argumento *time* se puede especificar en uno de los formatos aceptados para el tipo de datos **datetime** o como una variable local. No se pueden especificar fechas; por tanto, no se permite la parte de fecha del valor **datetime**.

TIME

Indica a SQL Server que espere hasta la hora especificada.

Observaciones

Después de ejecutar la instrucción WAITFOR, no puede utilizar la conexión a SQL Server hasta que llegue la hora o se produzca el suceso que ha especificado.

Para ver los procesos activos y en espera, utilice **sp_who**.

Ejemplos

A. Utilizar WAITFOR TIME

Este ejemplo ejecuta el procedimiento almacenado **update_all_stats** a las 10:20 p.m.

```
BEGIN
   WAITFOR TIME '22:20'
   EXECUTE update_all_stats
END
```

B. Utilizar WAITFOR DELAY

Este ejemplo muestra cómo se puede utilizar una variable local con la opción WAITFOR DELAY. Se crea un procedimiento almacenado de forma que espere una cantidad de tiempo variable y, a continuación, se devuelve al usuario la información como el número de horas, minutos y segundos que han transcurrido.

```
CREATE PROCEDURE time_delay @@DELAYLENGTH char(9)
AS
DECLARE @@RETURNINFO varchar(255)
BEGIN
   WAITFOR DELAY @@DELAYLENGTH
   SELECT @@RETURNINFO = 'A total time of ' +
                  SUBSTRING(@@DELAYLENGTH, 1, 3) +
                  ' hours, ' +
                  SUBSTRING(@@DELAYLENGTH, 5, 2) +
                  ' minutes, and ' +
                  SUBSTRING(@@DELAYLENGTH, 8, 2) +
                  ' seconds, ' +
                  'has elapsed! Your time is up.'
   PRINT @@RETURNINFO
END
GO
-- This next statement executes the time_delay procedure.
EXEC time_delay '000:00:10'
GO
```

El siguiente es el conjunto de resultados:

```
A total time of 000 hours, 00 minutes, and 10 seconds, has elapsed! Your
time is up.
```

WHILE

Establece una condición para la ejecución repetida de una instrucción o bloque de instrucciones de SQL. Las instrucciones se ejecutan repetidamente mientras la condición especificada sea verdadera. Se puede controlar la ejecución de instrucciones en el bucle WHILE con las palabras clave BREAK y CONTINUE.

Sintaxis

WHILE *Boolean_expression*
   {*sql_statement* | *statement_block*}

[BREAK]
{*sql_statement* | *statement_block*}
[CONTINUE]

Argumentos

Boolean_expression

Es una expresión que devuelve TRUE o FALSE. Si la expresión booleana contiene una instrucción SELECT, la instrucción SELECT debe ir entre paréntesis.

{*sql_statement* | *statement_block*}

Se trata de cualquier instrucción o grupo de instrucciones Transact-SQL definidos con un bloque de instrucciones. Para definir un bloque de instrucciones, utilice las palabras clave de control de flujo BEGIN y END.

BREAK

Hace que se salga del bloque WHILE más interno. Se ejecutan las instrucciones que aparecen después de la palabra clave END, que marca el final del bucle.

CONTINUE

Hace que se reinicie el bucle WHILE y omite las instrucciones que haya después de la palabra clave CONTINUE.

Observaciones

Si dos o más bucles WHILE están anidados, la instrucción BREAK interna sale al siguiente bucle más externo. Primero se ejecutan todas las instrucciones que haya después del final del bucle interno y, a continuación, se reinicia el siguiente bucle más externo.

Ejemplos

A. Utilizar BREAK y CONTINUE con IF...ELSE y WHILE anidados

En este ejemplo, si el promedio de precio es menor de 30 $, el bucle WHILE dobla los precios y, a continuación, selecciona el precio máximo. Si el precio máximo es menor o igual que 50 $, el bucle WHILE se reinicia y dobla los precios de nuevo. Este bucle continúa la duplicación de precios hasta que el precio máximo sea mayor que 50 $ y, a continuación, sale del bucle WHILE e imprime un mensaje.

```
USE pubs
GO
WHILE (SELECT AVG(price) FROM titles) < $30
BEGIN
```

```
   UPDATE titles
      SET price = price * 2
   SELECT MAX(price) FROM titles
   IF (SELECT MAX(price) FROM titles) > $50
      BREAK
   ELSE
      CONTINUE
END
PRINT 'Too much for the market to bear'
```

B. Utilizar WHILE en un procedimiento con cursores

La construcción WHILE siguiente es una sección de un procedimiento llamado **count_all_rows**. Para este ejemplo, esta construcción WHILE prueba el valor devuelto de @@FETCH_STATUS, una función que se utiliza con cursores. Debido a que @@FETCH_STATUS puede devolver -2, -1 ó 0, se deben probar los tres casos. Si se eliminó una fila de los resultados del cursor desde el momento en que se ejecutó este procedimiento almacenado, se omite esa fila. Una recuperación correcta (0) hace que se ejecute la instrucción SELECT del bucle BEGIN...END.

```
USE pubs
DECLARE tnames_cursor CURSOR
FOR
   SELECT TABLE_NAME
   FROM INFORMATION_SCHEMA.TABLES
OPEN tnames_cursor
DECLARE @tablename sysname
--SET @tablename = 'authors'
FETCH NEXT FROM tnames_cursor INTO @tablename
WHILE (@@FETCH_STATUS <> -1)
BEGIN
   IF (@@FETCH_STATUS <> -2)
   BEGIN
      SELECT @tablename = RTRIM(@tablename)
      EXEC ('SELECT ''' + @tablename + ''' = count(*) FROM '
            + @tablename )
      PRINT ' '
   END
   FETCH NEXT FROM tnames_cursor INTO @tablename
END
CLOSE tnames_cursor
DEALLOCATE tnames_cursor
```

CASE

Evalúa una lista de condiciones y devuelve como resultado una de las distintas expresiones posibles.

CASE tiene dos formatos:

- La función CASE sencilla compara una expresión con un conjunto de expresiones sencillas para determinar el resultado.
- La función CASE de búsqueda evalúa un conjunto de expresiones booleanas para determinar el resultado.

Ambos formatos aceptan el argumento ELSE opcional.

Sintaxis

Función CASE sencilla:

```
CASE input_expression
   WHEN when_expression THEN result_expression
      [ ...n ]
   [
      ELSE else_result_expression
   ]
END
```

Función CASE de búsqueda:

```
CASE
   WHEN Boolean_expression THEN result_expression
      [ ...n ]
   [
      ELSE else_result_expression
   ]
END
```

Argumentos

input_expression

Es la expresión que se evalúa cuando se utiliza el formato CASE sencillo. *input_expression* es cualquier expresión válida en Microsoft® SQL Server.™

WHEN *when_expression*

Es una expresión sencilla con la que se compara *input_expression* cuando se utiliza el formato CASE sencillo. *when_expression* es cualquier expresión válida en SQL Server. Los tipos de datos de *input_expression* y de cada *when_expression* tienen que ser los mismos o tiene que ser posible realizar una conversión implícita.

n

Es un marcador de posición que indica que se pueden utilizar varias cláusulas WHEN *when_expression* THEN *result_expression* o varias cláusulas WHEN *Boolean_expression* THEN *result_expression*.

THEN *result_expression*

Es la expresión que se devuelve cuando *input_expression* es igual a *when_expression* o cuando *Boolean_expression* es TRUE. *result_expression* es cualquier expresión válida en SQL Server.

ELSE *else_result_expression*

Es la expresión que se devuelve si ninguna comparación es igual a TRUE. Si se omite este argumento y ninguna comparación es igual a TRUE, CASE devuelve NULL. *else_result_expression* es cualquier expresión válida en SQL Server. Los tipos de datos de *else_result_expression* y cualquier *result_expression* tienen que ser los mismos o tiene que ser posible realizar una conversión implícita.

WHEN *Boolean_expression*

Es la expresión booleana que se evalúa cuando se utiliza el formato CASE de búsqueda. *Boolean_expression* es cualquier expresión booleana válida.

Tipos de resultado

El tipo de resultado será el de mayor precedencia entre los tipos de datos usados en las expresiones *result_expressions* y la expresión opcional *else_result_expression*.

Valores de resultado

Función CASE sencilla:

- Evalúa *input_expression* y después, en el orden especificado, evalúa *input_expression* = *when_expression* por cada cláusula WHEN.
- Devuelve el valor de la expresión *result_expression* de la primera comparación (*input_expression* = *when_expression*) cuyo resultado sea TRUE.
- Si ninguna comparación *input_expression* = *when_expression* da como resultado TRUE, SQL Server devuelve el valor de la expresión *else_result_expression*, si es que se especificó la cláusula ELSE, o un valor NULL en caso contrario.

Función CASE de búsqueda:

- Se evalúa, en el orden especificado, la expresión *Boolean_expression* de cada cláusula WHEN.

- Devuelve el valor de la expresión *result_expression* de la primera expresión *Boolean_expression* cuya evaluación dé como resultado TRUE.

- Si ninguna *Boolean_expression* es TRUE, SQL Server devuelve *else_result_expression*, si se especifica la cláusula ELSE, o un valor NULL si no se especifica la cláusula ELSE.

Ejemplos

A. Utilizar una instrucción SELECT con una función CASE sencilla

En una instrucción SELECT, una función CASE sencilla sólo permite una comprobación de igualdad; no se pueden hacer otras comparaciones. Este ejemplo utiliza la función CASE para modificar la presentación de una clasificación de libros con el fin de hacerla más comprensible.

```
USE pubs
GO
SELECT   Category =
      CASE type
         WHEN 'popular_comp' THEN 'Popular Computing'
         WHEN 'mod_cook' THEN 'Modern Cooking'
         WHEN 'business' THEN 'Business'
         WHEN 'psychology' THEN 'Psychology'
         WHEN 'trad_cook' THEN 'Traditional Cooking'
         ELSE 'Not yet categorized'
      END,
   CAST(title AS varchar(25)) AS 'Shortened Title',
   price AS Price
FROM titles
WHERE price IS NOT NULL
ORDER BY type, price
COMPUTE AVG(price) BY type
GO
```

El siguiente es el conjunto de resultados:

```
Category            Shortened Title           Price
------------------- ------------------------- -------------------------
Business            You Can Combat Computer S 2.99
Business            Cooking with Computers: S 11.95
Business            The Busy Executive's Data 19.99
Business            Straight Talk About Compu 19.99

                                              avg
                                              =========================
                                              13.73

Category            Shortened Title           Price
------------------- ------------------------- -------------------------
Modern Cooking      The Gourmet Microwave     2.99
Modern Cooking      Silicon Valley Gastronomi 19.99
```

```
                                          avg
                                          ==========================
                                          11.49

Category            Shortened Title           Price
------------------- ------------------------- --------------------------
Popular Computing   Secrets of Silicon Valley 20.00
Popular Computing   But Is It User Friendly?  22.95

                                          avg
                                          ==========================
                                          21.48

Category            Shortened Title           Price
------------------- ------------------------- --------------------------
Psychology          Life Without Fear         7.00
Psychology          Emotional Security: A New 7.99
Psychology          Is Anger the Enemy?       10.95
Psychology          Prolonged Data Deprivatio 19.99
Psychology          Computer Phobic AND Non-P 21.59

                                          avg
                                          ==========================
                                          13.50

Category            Shortened Title           Price
------------------- ------------------------- --------------------------
Traditional Cooking Fifty Years in Buckingham 11.95
Traditional Cooking Sushi, Anyone?            14.99
Traditional Cooking Onions, Leeks, and Garlic 20.95

                                          avg
                                          ==========================
                                          15.96

(21 row(s) affected)
```

B. Utilizar una instrucción SELECT con una función CASE sencilla y otra de búsqueda

En una instrucción SELECT, la función CASE de búsqueda permite sustituir valores en el conjunto de resultados basándose en los valores de comparación. Este ejemplo presenta el precio (una columna **money**) como comentario basado en el intervalo de precios de cada libro.

```
USE pubs
GO
SELECT    'Price Category' =
      CASE
         WHEN price IS NULL THEN 'Not yet priced'
         WHEN price < 10 THEN 'Very Reasonable Title'
         WHEN price >= 10 and price < 20 THEN 'Coffee Table Title'
         ELSE 'Expensive book!'
      END,
   CAST(title AS varchar(20)) AS 'Shortened Title'
```

```
FROM titles
ORDER BY price
GO
```

El siguiente es el conjunto de resultados:

```
Price Category        Shortened Title
--------------------- --------------------
Not yet priced        Net Etiquette
Not yet priced        The Psychology of Co
Very Reasonable Title The Gourmet Microwav
Very Reasonable Title You Can Combat Compu
Very Reasonable Title Life Without Fear
Very Reasonable Title Emotional Security:
Coffee Table Title    Is Anger the Enemy?
Coffee Table Title    Cooking with Compute
Coffee Table Title    Fifty Years in Bucki
Coffee Table Title    Sushi, Anyone?
Coffee Table Title    Prolonged Data Depri
Coffee Table Title    Silicon Valley Gastr
Coffee Table Title    Straight Talk About
Coffee Table Title    The Busy Executive's
Expensive book!       Secrets of Silicon V
Expensive book!       Onions, Leeks, and G
Expensive book!       Computer Phobic And
Expensive book!       But Is It User Frien

(18 row(s) affected)
```

C. Utilizar CASE con SUBSTRING y SELECT

Este ejemplo utiliza CASE y THEN para generar una lista de autores, los números de identificación de los libros y los tipos de libros que cada autor ha escrito.

```
USE pubs
SELECT SUBSTRING((RTRIM(a.au_fname) + ' '+
   RTRIM(a.au_lname) + ' '), 1, 25) AS Name, a.au_id, ta.title_id,
   Type =
  CASE
    WHEN SUBSTRING(ta.title_id, 1, 2) = 'BU' THEN 'Business'
    WHEN SUBSTRING(ta.title_id, 1, 2) = 'MC' THEN 'Modern Cooking'
    WHEN SUBSTRING(ta.title_id, 1, 2) = 'PC' THEN 'Popular Computing'
    WHEN SUBSTRING(ta.title_id, 1, 2) = 'PS' THEN 'Psychology'
    WHEN SUBSTRING(ta.title_id, 1, 2) = 'TC' THEN 'Traditional Cooking'
  END
FROM titleauthor ta JOIN authors a ON ta.au_id = a.au_id
```

El siguiente es el conjunto de resultados:

```
Name                      au_id       title_id Type
------------------------- ----------- -------- -------------------
Johnson White             172-32-1176 PS3333   Psychology
Marjorie Green            213-46-8915 BU1032   Business
Marjorie Green            213-46-8915 BU2075   Business
```

Cheryl Carson	238-95-7766	PC1035	Popular Computing
Michael O'Leary	267-41-2394	BU1111	Business
Michael O'Leary	267-41-2394	TC7777	Traditional Cooking
Dean Straight	274-80-9391	BU7832	Business
Abraham Bennet	409-56-7008	BU1032	Business
Ann Dull	427-17-2319	PC8888	Popular Computing
Burt Gringlesby	472-27-2349	TC7777	Traditional Cooking
Charlene Locksley	486-29-1786	PC9999	Popular Computing
Charlene Locksley	486-29-1786	PS7777	Psychology
Reginald Blotchet-Halls	648-92-1872	TC4203	Traditional Cooking
Akiko Yokomoto	672-71-3249	TC7777	Traditional Cooking
Innes del Castillo	712-45-1867	MC2222	Modern Cooking
Michel DeFrance	722-51-5454	MC3021	Modern Cooking
Stearns MacFeather	724-80-9391	BU1111	Business
Stearns MacFeather	724-80-9391	PS1372	Psychology
Livia Karsen	756-30-7391	PS1372	Psychology
Sylvia Panteley	807-91-6654	TC3218	Traditional Cooking
Sheryl Hunter	846-92-7186	PC8888	Popular Computing
Anne Ringer	899-46-2035	MC3021	Modern Cooking
Anne Ringer	899-46-2035	PS2091	Psychology
Albert Ringer	998-72-3567	PS2091	Psychology
Albert Ringer	998-72-3567	PS2106	Psychology

(25 row(s) affected)

DECLARACIONES

DECLARE @local_variable

Las variables se declaran en el cuerpo de un proceso por lotes o procedimiento con la instrucción DECLARE, y se les asignan valores con una instrucción SET o SELECT. Las variables de cursor pueden declararse con esta instrucción y utilizarse con otras instrucciones relacionadas con los cursores. Después de su declaración, todas las variables se inicializan con NULL (nulo).

Sintaxis

```
DECLARE
   {{ @local_variable data_type }
      | { @cursor_variable_name CURSOR }
      | { table_type_definition }
   } [ ,...n]

< table_type_definition > ::=
   TABLE ( { < column_definition > | < table_constraint > } [ ,... ]
         )

< column_definition > ::=
   column_name scalar_data_type
   [ COLLATE collation_name ]
   [ [ DEFAULT constant_expression ] | IDENTITY [ ( seed, increment ) ] ]
   [ ROWGUIDCOL ]
   [ < column_constraint > ]

< column_constraint > ::=
   { [ NULL | NOT NULL ]
   | [ PRIMARY KEY | UNIQUE ]
   | CHECK ( logical_expression )
   }

< table_constraint > ::=
   { { PRIMARY KEY | UNIQUE } ( column_name [ ,... ] )
   | CHECK ( search_condition )
   }
```

Argumentos

@*local_variable*

Es el nombre de una variable. Los nombres de variables deben comenzar por un signo de arroba (**@**). Los nombres de variables locales deben seguir las reglas de los identificadores.

data_type

Se trata de un tipo de datos proporcionado por el sistema o definido por el usuario. El tipo de datos de una variable no puede ser **text**, **ntext** ni **image**.

@*cursor_variable_name*

Es el nombre de una variable de cursor. Los nombres de variable de cursor deben comenzar con un signo de arroba (**@**) y seguir las reglas de los identificadores.

CURSOR

Especifica que la variable es una variable de cursor local.

table_type_definition

Define el tipo de datos de tabla. La declaración de tabla incluye definiciones de columna, nombres, tipos de datos y restricciones. Sólo se permiten los tipos de restricciones PRIMARY KEY, UNIQUE KEY, NULL y CHECK.

table_type_definition es un subconjunto de información que se utiliza para definir una tabla en CREATE TABLE. Aquí se incluyen los elementos y definiciones fundamentales.

n

Es un marcador de posición que indica que se pueden especificar y asignar valores a varias variables. Cuando se declara una variable de tabla, ésta debe ser la única variable que se declara en la instrucción DECLARE.

column_name

Es el nombre de la columna de la tabla.

scalar_data_type

Especifica que la columna es de un tipo de datos escalar.

[COLLATE *collation_name*]

Especifica la intercalación de la columna; *collation_name* puede ser un nombre de intercalación de Windows o de SQL, y sólo se aplica a las columnas de tipos de datos **char**, **varchar**, **text**, **nchar**, **nvarchar** y **ntext**. Si no se especifica, se asignará a la columna la

intercalación del tipo de datos definido por el usuario, si la columna es de un tipo de datos definido por el usuario, o la intercalación predeterminada de la base de datos.

DEFAULT

Especifica el valor suministrado para la columna cuando no se ha especificado explícitamente un valor durante la inserción. Las definiciones DEFAULT se pueden aplicar a cualquier columna, excepto a las definidas como **timestamp** o aquellas que tengan la propiedad IDENTITY. Las definiciones DEFAULT se quitan cuando se quita la tabla. Como valor predeterminado sólo se puede utilizar un valor constante, por ejemplo una cadena de caracteres, una función del sistema, como SYSTEM_USER() o NULL. Para mantener la compatibilidad con las versiones anteriores de SQL Server, se puede asignar un nombre de restricción a DEFAULT.

constant_expression

Es una constante, NULL o una función del sistema utilizados como el valor predeterminado de una columna.

IDENTITY

Indica que la nueva columna es una columna identidad. Cuando se agrega una nueva fila a la tabla, SQL Server proporciona un valor incremental y único para la columna. Las columnas de identidad se utilizan normalmente junto con restricciones PRIMARY KEY para que actúen como identificador exclusivo de fila para la tabla. La propiedad IDENTITY puede asignarse a las columnas **tinyint**, **smallint**, **int**, **decimal(p,0)** o **numeric(p,0)**. Sólo se puede crear una columna de identidad por tabla. Los valores predeterminados enlazados y las restricciones DEFAULT no se pueden utilizar con una columna identidad. Es necesario especificar la inicialización y el incremento, o ninguno de los dos. Si no se especifica ninguno, el valor predeterminado es (1,1).

seed

Es el valor que se utiliza para la primera fila cargada en la tabla.

increment

Se trata del valor incremental que se agrega al valor de identidad de la anterior fila cargada.

ROWGUIDCOL

Indica que la nueva columna es una columna de identificador exclusivo global de fila. Únicamente se puede designar una columna **uniqueidentifier** por cada tabla como la columna ROWGUIDCOL. La propiedad ROWGUIDCOL se puede asignar únicamente a una columna **uniqueidentifier**.

NULL | NOT NULL

Son palabras clave que determinan si se permiten o no valores Null en la columna.

PRIMARY KEY

Es una restricción que exige la integridad de entidad para una o varias columnas dadas a través de un índice único. Sólo se puede crear una restricción PRIMARY KEY por cada tabla.

UNIQUE

Es una restricción que proporciona la integridad de entidad para una o varias columnas dadas a través de un índice único. Una tabla puede tener varias restricciones UNIQUE.

CHECK

Es una restricción que exige la integridad del dominio al limitar los valores posibles que se pueden escribir en una o varias columnas.

logical_expression

Es una expresión lógica que devuelve TRUE o FALSE.

Observaciones

Las variables se suelen utilizar en un proceso por lotes o procedimiento como contadores para WHILE, LOOP o un bloque IF...ELSE.

Las variables sólo se pueden utilizar en expresiones y no en lugar de nombres de objeto o palabras clave. Para formar instrucciones SQL dinámicas, utilice EXECUTE.

El alcance de una variable local es el proceso por lotes, procedimiento almacenado o bloque de instrucciones en que se declaró.

Se puede hacer referencia como origen a una variable de cursor que actualmente tiene asignado un cursor en una instrucción:

- CLOSE.
- DEALLOCATE.
- FETCH.
- OPEN.
- DELETE o UPDATE por posición.

- SET CURSOR variable (en el lado derecho).

En todas estas instrucciones, Microsoft® SQL Server™ genera un error si la variable de cursor a la que se hace referencia existe pero actualmente no tiene asignado un cursor. Si una variable de cursor a la que se hace referencia no existe, SQL Server genera el mismo error que genera para una variable no declarada de otro tipo.

Una variable de cursor:

- Puede ser el destino de un tipo de cursor u otra variable de cursor.
- Se puede hacer referencia a la variable de cursor como el destino de un parámetro de cursor de resultado en una instrucción EXECUTE si la variable de cursor no tiene actualmente un cursor asignado.
- Se debe considerar como un puntero al cursor.

Ejemplos

A. Utilizar DECLARE

Este ejemplo utiliza una variable local denominada **@find** para recuperar información de todos los autores cuyos apellidos comienzan con Ring.

```
USE pubs
DECLARE @find varchar(30)
SET @find = 'Ring%'
SELECT au_lname, au_fname, phone
FROM authors
WHERE au_lname LIKE @find
```

El siguiente es el conjunto de resultados:

```
au_lname                                 au_fname             phone
---------------------------------------- -------------------- ------------
Ringer                                   Anne                 801 826-0752
Ringer                                   Albert               801 826-0752

(2 row(s) affected)
```

B. Utilizar DECLARE con dos variables

Este ejemplo recupera nombres de empleados de los empleados de Binnet & Hardley (**pub_id** = 0877) contratados el 1 de enero de 1993 o posteriormente.

```
USE pubs
SET NOCOUNT ON
GO
```

```
DECLARE @pub_id char(4), @hire_date datetime
SET @pub_id = '0877'
SET @hire_date = '1/01/93'
-- Here is the SELECT statement syntax to assign values to two local
-- variables.
-- SELECT @pub_id = '0877', @hire_date = '1/01/93'
SET NOCOUNT OFF
SELECT fname, lname
FROM employee
WHERE pub_id = @pub_id and hire_date >= @hire_date
```

El siguiente es el conjunto de resultados:

```
fname                lname
-------------------- ------------------------------
Anabela              Domingues
Paul                 Henriot

(2 row(s) affected)
```

MANEJO DE ERRORES

@@ERROR

Devuelve el número de error de la última instrucción Transact-SQL ejecutada.

Sintaxis

@@ERROR

Tipos devueltos

integer

Observaciones

Cuando Microsoft® SQL Server™ completa con éxito la ejecución de una instrucción Transact-SQL, en @@ERROR se establece el valor 0. Si se produce un error, se devuelve un mensaje de error. @@ERROR devuelve el número del mensaje de error, hasta que se ejecute otra instrucción Transact-SQL. Puede ver el texto asociado a un número de error @@ERROR en la tabla de sistema **sysmessages**.

Al restablecerse @@ERROR con cada instrucción ejecutada, debe comprobarlo inmediatamente después de la instrucción que desea validar o guardarlo en una variable local para examinarlo posteriormente.

Ejemplos

A. Utilizar @@ERROR para detectar un error específico

En este ejemplo se utiliza @@ERROR para comprobar si se infringe una restricción CHECK (error nº 547) en una instrucción UPDATE.

```
USE pubs
GO
UPDATE authors SET au_id = '172 32 1176'
WHERE au_id = "172-32-1176"

IF @@ERROR = 547
   print "A check constraint violation occurred"
```

B. Utilizar @@ERROR para salir condicionalmente de un procedimiento

La instrucción IF...ELSE de este ejemplo comprueba @@ERROR después de una instrucción INSERT de un procedimiento almacenado. El valor de la variable @@ERROR

determina el código de retorno enviado al programa que llamó, lo que indica el éxito o el fracaso del procedimiento.

```
USE pubs
GO

-- Create the procedure.
CREATE PROCEDURE add_author
@au_id varchar(11),@au_lname varchar(40),
@au_fname varchar(20),@phone char(12),
@address varchar(40) = NULL,@city varchar(20) = NULL,
@state char(2) = NULL,@zip char(5) = NULL,
@contract bit = NULL
AS

-- Execute the INSERT statement.
INSERT INTO authors
(au_id,  au_lname, au_fname, phone, address,
 city, state, zip, contract) values
(@au_id,@au_lname,@au_fname,@phone,@address,
 @city,@state,@zip,@contract)

-- Test the error value.
IF @@ERROR <> 0
BEGIN
   -- Return 99 to the calling program to indicate failure.
   PRINT "An error occurred loading the new author information"
   RETURN(99)
END
ELSE
BEGIN
   -- Return 0 to the calling program to indicate success.
   PRINT "The new author information has been loaded"
   RETURN(0)
END
GO
```

C. Utilizar @@ERROR para comprobar el éxito de varias instrucciones

Este ejemplo depende de la ejecución con éxito de las instrucciones INSERT y DELETE. Se establece el valor de @@ERROR en variables locales después de ambas instrucciones y se utilizan las variables en una rutina de tratamiento de errores común para la operación.

```
USE pubs
GO
DECLARE @del_error int, @ins_error int
-- Start a transaction.
BEGIN TRAN

-- Execute the DELETE statement.
DELETE authors
WHERE au_id = '409-56-7088'

-- Set a variable to the error value for
-- the DELETE statement.
```

```
SELECT @del_error = @@ERROR

-- Execute the INSERT statement.
INSERT authors
   VALUES('409-56-7008', 'Bennet', 'Abraham', '415 658-9932',
   '6223 Bateman St.', 'Berkeley', 'CA', '94705', 1)
-- Set a variable to the error value for
-- the INSERT statement.
SELECT @ins_error = @@ERROR

-- Test the error values.
IF @del_error = 0 AND @ins_error = 0
BEGIN
   -- Success. Commit the transaction.
   PRINT "The author information has been replaced"
   COMMIT TRAN
END
ELSE
BEGIN
   -- An error occurred. Indicate which operation(s) failed
   -- and roll back the transaction.
   IF @del_error <> 0
      PRINT "An error occurred during execution of the DELETE
      statement."

   IF @ins_error <> 0
      PRINT "An error occurred during execution of the INSERT
      statement."

   ROLLBACK TRAN
END
GO
```

D. Utilizar @@ERROR con @@ROWCOUNT

Este ejemplo utiliza @@ERROR con @@ROWCOUNT para validar la operación de una instrucción UPDATE. Se comprueba el valor de @@ERROR para ver si hay un error y se utiliza @@ROWCOUNT para asegurar que la actualización se aplica correctamente a una fila de la tabla.

```
USE pubs
GO
CREATE PROCEDURE change_publisher
@title_id tid,
@new_pub_id char(4)
AS

-- Declare variables used in error checking.
DECLARE @error_var int, @rowcount_var int

-- Execute the UPDATE statement.
UPDATE titles SET pub_id = @new_pub_id
WHERE title_id = @title_id

-- Save the @@ERROR and @@ROWCOUNT values in local
```

```
-- variables before they are cleared.
SELECT @error_var = @@ERROR, @rowcount_var = @@ROWCOUNT

-- Check for errors. If an invalid @new_pub_id was specified
-- the UPDATE statement returns a foreign-key violation error #547.
IF @error_var <> 0
BEGIN
   IF @error_var = 547
   BEGIN
      PRINT "ERROR: Invalid ID specified for new publisher"
      RETURN(1)
   END
   ELSE
   BEGIN
      PRINT "ERROR: Unhandled error occurred"
      RETURN(2)
   END
END

-- Check the rowcount. @rowcount_var is set to 0
-- if an invalid @title_id was specified.
IF @rowcount_var = 0
BEGIN
   PRINT "Warning: The title_id specified is not valid"
   RETURN(1)
END
ELSE
BEGIN
   PRINT "The book has been updated with the new publisher"
   RETURN(0)
END
GO
```

RAISERROR

Devuelve un mensaje de error definido por el usuario y establece un indicador del sistema para registrar que se ha producido un error. Con RAISERROR, el cliente puede obtener una entrada de la tabla **sysmessages** o generar un mensaje dinámicamente con una gravedad definida por el usuario y con información acerca del estado. Una vez definido, el mensaje se devuelve al cliente como un mensaje de error del servidor.

Sintaxis

RAISERROR **(** { *msg_id* | *msg_str* } { **,** *severity* **,** *state* }
[**,** *argument* [**,**...*n*]] **)**
[WITH *option* [**,**...*n*]]

Argumentos

msg_id

Se trata de un mensaje de error definido por el usuario que está almacenado en la tabla **sysmessages**. Los números de error de los mensajes definidos por el usuario deben ser mayores de 50.000. Los mensajes ad hoc generan el error 50.000.

msg_str

Se trata de un mensaje ad hoc con un formato similar al estilo de formato PRINTF que se utiliza en C. El mensaje de error puede contener un máximo de 400 caracteres. Si el mensaje contiene más de 400 caracteres, solamente aparecerán los 397 primeros y se agregarán puntos suspensivos para indicar que el mensaje está cortado. Todos los mensajes ad hoc tienen un identificador estándar de 14.000.

Para *msg_str* se admite este formato:

% [[*flag*] [*width*] [*precision*] [{h | l}]] *type*

Los parámetros que se pueden utilizar en *msg_str* son:

flag

Es un código que determina el espaciado y la justificación del mensaje de error definido por el usuario.

Código	Prefijo o justificación	Descripción
- (menos)	Justificado a la izquierda	Justifica a la izquierda el resultado dentro del ancho de campo dado.
+ (más)	Prefijo + (más) o - (menos)	Coloca el signo más (+) o menos (-) delante del valor de salida, si éste es del tipo con signo.
0 (cero)	Relleno con ceros	Si el ancho lleva 0 delante, se agregan ceros hasta que se alcanza el ancho mínimo. Cuando aparecen 0 y -, el 0 se omite. Cuando se especifica 0 con un formato entero (i, u, x, X, o, d), el 0 se omite.
# (número)	Prefijo 0x del tipo hexadecimal x o X	Cuando se utiliza con el formato o, x o X, el indicador # se coloca delante de los valores distintos de cero con 0, 0x o 0X, respectivamente. Cuando d, i, o u llevan delante el indicador #, se omite el indicador.
' ' (blanco)	Relleno con espacios	Coloca delante del valor de salida espacios en blanco si el valor tiene signo y es positivo. Se omite cuando

		se incluye con el indicador más (+).

width

Es un entero que define el ancho mínimo. Con un asterisco (*) se permite utilizar *precision* para determinar el ancho.

precision

Es el número máximo de caracteres impresos en el campo de salida o el número mínimo de dígitos impresos para los valores enteros. Con un asterisco (*) se permite utilizar *argument* para determinar la precisión.

{h | l} *type*

Se utiliza con los tipos de caracteres d, i, o, x, X ó u, y crea valores **short int** (h) o **long int** (l).

Tipo de carácter	Representa
d o I	Entero con signo.
o	Octal sin signo.
p	Puntero
s	String
u	Entero sin signo.
x o X	Hexadecimal sin signo

Nota Los tipos de datos **float** de dos y un carácter no se admiten.

severity

Se trata del nivel de gravedad definido por el usuario que se asocia con este mensaje. Todos los usuarios pueden utilizar los niveles de gravedad de 0 a 18. Sólo los miembros de la

función fija de servidor **sysadmin** pueden utilizar los niveles de gravedad de 19 a 25. Para los niveles de gravedad de 19 a 25, se necesita la opción WITH LOG.

> **Advertencia** Los niveles de gravedad de 20 a 25 se consideran muy negativos. Si se encuentra un nivel de gravedad de este tipo, la conexión cliente termina tras recibir el mensaje. El error se incluye en el registro de errores y en el registro de la aplicación.

state

Es un entero arbitrario entre 1 y 127 que representa información acerca del estado de llamada del error. Un valor negativo de *state* pasa a tener un valor predeterminado de 1.

argument

Se trata de los parámetros utilizados para sustituir las variables definidas en *msg_str* o en los mensajes que corresponden a *msg_id*. Puede haber cero o más parámetros de sustitución; sin embargo, el número total de parámetros de sustitución no puede ser más de veinte. Cada parámetro de sustitución puede ser una variable local o uno de estos tipos de datos: **int1**, **int2**, **int4**, **char**, **varchar**, **binary** o **varbinary**. No se admiten otros tipos de datos.

option

Se trata de una opción personalizada del error. *option* puede ser uno de estos valores.

Valor	Descripción
LOG	Guarda el error en el registro de errores del servidor y en el registro de la aplicación. Los errores guardados en el registro de errores del servidor están limitados actualmente a un máximo de 440 bytes.
NOWAIT	Envía inmediatamente los mensajes al cliente.
SETERROR	Establece el valor **@@**ERROR a *msg_id* ó a 50000, independientemente del nivel de gravedad.

Observaciones

Si se utiliza un error de **sysmessages** y el mensaje se creó con el formato mostrado para *msg_str*, los argumentos proporcionados (*argument1*, *argument2*, etc.) se pasan al mensaje del *msg_id* proporcionado.

Cuando utilice RAISERROR para crear y devolver mensajes de error definidos por el usuario, utilice **sp_addmessage** para agregar este tipo de mensajes de error y **sp_dropmessage** para eliminarlos.

Cuando se genera un error, su número se coloca en la función @@ERROR, que almacena los números de error generados más recientemente. De forma predeterminada, @@ERROR se establece en 0 para los mensajes que tienen una gravedad de 1 a 10.

Ejemplos

A. Crear un mensaje ad hoc

En este ejemplo se muestran dos errores posibles. El primero es un error simple que contiene un mensaje estático. El segundo error se genera dinámicamente en función de la modificación que se haya intentado realizar.

```
CREATE TRIGGER employee_insupd
ON employee
FOR INSERT, UPDATE
AS
/* Get the range of level for this job type from the jobs table. */
DECLARE @@MIN_LVL tinyint,
   @@MAX_LVL tinyint,
   @@EMP_LVL tinyint,
   @@JOB_ID smallint
SELECT @@MIN_LVl = min_lvl,
   @@MAX_LV = max_lvl,
   @@ EMP_LVL = i.job_lvl,
   @@JOB_ID = i.job_id
FROM employee e, jobs j, inserted i
WHERE e.emp_id = i.emp_id AND i.job_id = j.job_id
IF (@@JOB_ID = 1) and (@@EMP_lVl <> 10)
BEGIN
   RAISERROR ('Job id 1 expects the default level of 10.', 16, 1)
   ROLLBACK TRANSACTION
END
ELSE
IF NOT @@ EMP_LVL BETWEEN @@MIN_LVL AND @@MAX_LVL)
BEGIN
   RAISERROR ('The level for job_id:%d should be between %d and %d.',
      16, 1, @@JOB_ID, @@MIN_LVL, @@MAX_LVL)
   ROLLBACK TRANSACTION
END
```

B. Crear un mensaje ad hoc en sysmessages

En este ejemplo se muestra cómo conseguir los mismos resultados con RAISERROR mediante el paso de parámetros a un mensaje almacenado en la tabla **sysmessages** al ejecutar el desencadenador **employee_insupd**. El mensaje se agregó a la tabla **sysmessages** con el procedimiento almacenado de sistema **sp_addmessage** como el número de mensaje 50005.

Nota Este ejemplo sólo se muestra con propósitos ilustrativos.

```
RAISERROR (50005, 16, 1, @@JOB_ID, @@MIN_LVL, @@MAX_LVL)
```

INSERCION DE COMENTARIOS

/*...*/ (comentario)

Indica texto proporcionado por el usuario. El servidor no evalúa el texto que hay entre los caracteres de comentarios **/*** y ***/**.

Sintaxis

/ * *text_of_comment* ***** **/**

Argumentos

text_of_comment

Es la cadena o cadenas de caracteres que contienen el texto del comentario.

Observaciones

Los comentarios se pueden insertar en una línea separada o dentro de una instrucción de Transact-SQL. Los comentarios con varias líneas deben indicarse con **/*** y ***/**. Una regla de estilo que se utiliza a menudo para los comentarios de varias líneas es comenzar la primera línea con /*, las siguientes con ** y finalizar con */.

No hay límite de longitud para los comentarios.

> **Nota** Si se incluye un comando GO en un comentario, se producirá un mensaje de error.

Ejemplos

En este ejemplo se utilizan comentarios para documentar y probar el comportamiento durante las diferentes fases del desarrollo de un desencadenador. En este ejemplo, las partes del desencadenador se marcan como comentario para reducir los problemas y probar sólo una de las condiciones. Se utilizan ambos tipos de caracteres de comentario; los comentarios con el estilo de SQL-92 (--) se muestran solos y anidados.

> **Nota** La siguiente instrucción CREATE TRIGGER no se ejecuta correctamente porque ya hay un desencadenador llamado **employee_insupd** en la base de datos **pubs**.

```
CREATE TRIGGER employee_insupd
/*
   Because CHECK constraints can only reference the column(s)
   on which the column- or table-level constraint has
```

```
   been defined, any cross-table constraints (in this case,
   business rules) need to be defined as triggers.

   Employee job_lvls (on which salaries are based) should be within
   the range defined for their job. To get the appropriate range,
   the jobs table needs to be referenced. This trigger will be
   invoked for INSERT and UPDATES only.
*/
ON employee
FOR INSERT, UPDATE
AS
/* Get the range of level for this job type from the jobs table. */
DECLARE @min_lvl tinyint,      -- Minimum level var. declaration
   @max_lvl tinyint,         -- Maximum level var. declaration
   @emp_lvl tinyint,         -- Employee level var. declaration
   @job_id smallint            -- Job ID var. declaration
SELECT @min_lvl = min_lvl,      -- Set the minimum level
   @max_lvl = max_lvl,          -- Set the maximum level
   @emp_lvl = i.job_lvl,      -- Set the proposed employee level
   @job_id = i.job_id         -- Set the Job ID for comparison
FROM employee e, jobs j, inserted i
WHERE e.emp_id = i.emp_id AND i.job_id = j.job_id
IF (@job_id = 1) and (@emp_lvl <> 10)
BEGIN
   RAISERROR ('Job id 1 expects the default level of 10.', 16, 1)
   ROLLBACK TRANSACTION
END
/* Only want to test first condition. Remaining ELSE is commented out.
-- Comments within this section are unaffected by this commenting style.
ELSE
IF NOT (@emp_lvl BETWEEN @min_lvl AND @max_lvl) -- Check valid range
BEGIN
   RAISERROR ('The level for job_id:%d should be between %d and %d.',
      16, 1, @job_id, @min_lvl, @max_lvl)
   ROLLBACK TRANSACTION
END
*/
GO
```

--(comentario)

Indica texto proporcionado por el usuario. Los comentarios se pueden insertar en una línea separada, anidada (sólo --) al final de una línea de comandos de Transact-SQL, o dentro de una instrucción de Transact-SQL. El servidor no evalúa el comentario. El indicador estándar de SQL-92 para los comentarios son dos guiones (--).

Sintaxis

-- *text_of_comment*

Argumentos

text_of_comment

Es la cadena de caracteres que contiene el texto del comentario.

Observaciones

Utilice los dos guiones (--) para comentarios de una línea o anidados. Los comentarios que se insertan con dos guiones (--) se delimitan con un carácter de nueva línea.

No hay límite de longitud para los comentarios.

> **Nota** Si se incluye un comando GO en un comentario, se producirá un mensaje de error.

Ejemplos

En este ejemplo se utilizan los caracteres -— de comentario.

```
-- Choose the pubs database.
USE pubs
-- Choose all columns and all rows from the titles table.
SELECT *
FROM titles
ORDER BY title_id ASC -- We don't have to specify ASC because that
-- is the default.
```

DESENCADENADORES

CREATE TRIGGER

Crea un desencadenador, que es una clase especial de procedimiento almacenado que se ejecuta automáticamente cuando un usuario intenta la instrucción especificada de modificación de datos en la tabla indicada. Microsoft® SQL Server™ permite crear varios desencadenadores para cualquier instrucción INSERT, UPDATE o DELETE.

Sintaxis

```
CREATE TRIGGER trigger_name
ON { table | view }
[ WITH ENCRYPTION ]
{
   { ( FOR | AFTER | INSTEAD OF ) { [ INSERT ] [ , ] [ UPDATE ] }
      [ WITH APPEND ]
      [ NOT FOR REPLICATION ]
      AS
      [ { IF UPDATE ( column )
         [ { AND | OR } UPDATE ( column ) ]
            [ ...n ]
      | IF ( COLUMNS_UPDATED ( ) { bitwise_operator } updated_bitmask )
            { comparison_operator } column_bitmask [ ...n ]
      } ]
      sql_statement [ ...n ]
   }
}
```

Argumentos

trigger_name

Es el nombre del desencadenador. Un nombre de desencadenador debe cumplir las reglas de los identificadores y debe ser único en la base de datos. Especificar el propietario del desencadenador es opcional.

Table | view

Es la tabla o vista en que se ejecuta el desencadenador; algunas veces se denomina tabla del desencadenador o vista del desencadenador. Especificar el nombre del propietario de la tabla o vista es opcional.

WITH ENCRYPTION

Codifica las entradas **syscomments** que contienen el texto de CREATE TRIGGER. El uso de WITH ENCRYPTION impide que el desencadenador se publique como parte de la duplicación de SQL Server.

AFTER

Especifica que el desencadenador sólo se activa cuando todas las operaciones especificadas en la instrucción SQL desencadenadora se han ejecutado correctamente. Además, todas las acciones referenciales en cascada y las comprobaciones de restricciones deben ser correctas para que este desencadenador se ejecute.

AFTER es el valor predeterminado, si sólo se especifica la palabra clave FOR.

Los desencadenadores AFTER no se pueden definir en las vistas.

INSTEAD OF

Especifica que se ejecuta el desencadenador *en vez de* la instrucción SQL desencadenadora, por lo que se suplantan las acciones de las instrucciones desencadenadoras.

Como máximo, se puede definir un desencadenador INSTEAD OF por cada instrucción INSERT, UPDATE o DELETE en cada tabla o vista. No obstante, en las vistas es posible definir otras vistas que tengan su propio desencadenador INSTEAD OF.

Los desencadenadores INSTEAD OF no se permiten en las vistas actualizables WITH CHECK OPTION. SQL Server emitirá un error si se agrega un desencadenador INSTEAD OF a una vista actualizable donde se ha especificado WITH CHECK OPTION. El usuario debe quitar esta opción mediante ALTER VIEW antes de definir el desencadenador INSTEAD OF.

{ [DELETE] [**,**] [INSERT] [**,**] [UPDATE] }

Son palabras clave que especifican qué instrucciones de modificación de datos activan el desencadenador cuando se intentan en esta tabla o vista. Se debe especificar al menos una opción. En la definición del desencadenador se permite cualquier combinación de éstas, en cualquier orden. Si especifica más de una opción, sepárelas con comas.

Para los desencadenadores INSTEAD OF, no se permite la opción DELETE en tablas que tengan una relación de integridad referencial que especifica una acción ON DELETE en cascada. Igualmente, no se permite la opción UPDATE en tablas que tengan una relación de integridad referencial que especifica una acción ON UPDATE en cascada.

WITH APPEND

Especifica que debe agregarse un desencadenador adicional de un tipo existente. La utilización de esta cláusula opcional sólo es necesaria cuando el nivel de compatibilidad es

65 o inferior. Si el nivel de compatibilidad es 70 o superior, no es necesaria la cláusula WITH APPEND para agregar un desencadenador adicional de un tipo existente (éste es el comportamiento predeterminado de CREATE TRIGGER cuando el nivel de compatibilidad es 70 o superior).

WITH APPEND no se puede utilizar con desencadenadores INSTEAD OF o cuando se ha declarado AFTER explícitamente. WITH APPEND sólo se puede utilizar si se especificó FOR (sin INSTEAD OF ni AFTER) por motivos de compatibilidad con versiones anteriores. WITH APPEND y FOR (que se interpreta como AFTER) no se admitirán en futuras versiones.

NOT FOR REPLICATION

Indica que el desencadenador no debe ejecutarse cuando un proceso de duplicación modifica la tabla involucrada en el mismo.

AS

Son las acciones que va a llevar a cabo el desencadenador.

sql_statement

Son las condiciones y acciones del desencadenador. Las condiciones del desencadenador especifican los criterios adicionales que determinan si los intentos de las instrucciones DELETE, INSERT o UPDATE hacen que se lleven a cabo las acciones del desencadenador.

Las acciones del desencadenador especificadas en las instrucciones Transact-SQL entran en efecto cuando se intenta la operación DELETE, INSERT o UPDATE.

Los desencadenadores pueden incluir cualquier número y clase de instrucciones Transact-SQL. Un desencadenador está diseñado para comprobar o cambiar los datos en base a una instrucción de modificación de datos; no debe devolver datos al usuario. Las instrucciones Transact-SQL de un desencadenador incluyen a menudo lenguaje de control de flujo. En las instrucciones CREATE TRIGGER se utilizan unas cuantas tablas especiales:

- **deleted** e **inserted** son tablas lógicas (conceptuales). Son de estructura similar a la tabla en que se define el desencadenador, es decir, la tabla en que se intenta la acción del usuario, y guarda los valores antiguos o nuevos de las filas que la acción del usuario puede cambiar. Por ejemplo, para recuperar todos los valores de la tabla **deleted**, utilice:
- SELECT *
- FROM deleted
- En un desencadenador DELETE, INSERT o UPDATE, SQL Server no admite referencias de columnas **text**, **ntext** o **image** en las tablas **inserted** y **deleted** si el nivel de compatibilidad es igual a 70. No se puede tener acceso a los valores **text**,

ntext e **image** de las tablas **inserted** y **deleted**. Para recuperar el nuevo valor de un desencadenador INSERT o UPDATE, combine la tabla **inserted** con la tabla de actualización original. Cuando el nivel de compatibilidad es 65 o inferior, se devuelven valores NULL para las columnas **inserted** o **deleted text**, **ntext** o **image** que admiten valores NULL; si las columnas no permiten valores NULL, se devuelven cadenas de longitud cero.

Si el nivel de compatibilidad es 80 o superior, SQL Server permite actualizar las columnas **text**, **ntext** o **image** mediante el desencadenador INSTEAD OF en tablas o vistas.

n

Se trata de un marcador de posición que indica que se pueden incluir varias instrucciones Transact-SQL en el desencadenador. Para la instrucción IF UPDATE **(***column***)**, se pueden incluir varias columnas repitiendo la cláusula UPDATE **(***column***)**.

IF UPDATE **(***column***)**

Prueba una acción INSERT o UPDATE en una columna especificada y no se utiliza con operaciones DELETE. Se puede especificar más de una columna. Como el nombre de la tabla se especifica en la cláusula ON, no lo incluya antes del nombre de la columna en una cláusula IF UPDATE. Para probar una acción INSERT o UPDATE para más de una columna, especifique una cláusula UPDATE**(***column***)** separada a continuación de la primera. IF UPDATE devolverá el valor TRUE en las acciones INSERT porque en las columnas se insertaron valores explícitos o implícitos (NULL).

Nota La cláusula IF UPDATE **(***column***)** funciona de forma idéntica a una instrucción IF, IF...ELSE o WHILE, y puede utilizar el bloque BEGIN...END.

UPDATE(*column*) puede utilizarse en cualquier parte dentro del cuerpo del desencadenador.

column

Es el nombre de la columna que se va a probar para una acción INSERT o UPDATE. Esta columna puede ser de cualquier tipo de datos admitido por SQL Server. No obstante, no se pueden utilizar columnas calculadas en este contexto.

IF **(**COLUMNS_UPDATED**())**

Prueba, sólo en un desencadenador INSERT o UPDATE, si la columna o columnas mencionadas se insertan o se actualizan. COLUMNS_UPDATED devuelve un patrón de bits **varbinary** que indica qué columnas de la tabla se insertaron o se actualizaron.

La función COLUMNS_UPDATED devuelve los bits en orden de izquierda a derecha, siendo el bit menos significativo el primero de la izquierda. El primer bit de la izquierda representa la primera columna de la tabla, el siguiente representa la segunda columna, etc. COLUMNS_UPDATED devuelve varios bytes si la tabla en que se ha creado el desencadenador contiene más de 8 columnas, siendo el menos significativo el primero de la izquierda. COLUMNS_UPDATED devolverá el valor TRUE en todas las columnas de las acciones INSERT porque en las columnas se insertaron valores explícitos o implícitos (NULL).

COLUMNS_UPDATED puede utilizarse en cualquier parte dentro del cuerpo del desencadenador.

bitwise_operator

Es el operador de bits que se utilizará en las comparaciones.

updated_bitmask

Es la máscara de bits de enteros de las columnas realmente actualizadas o insertadas. Por ejemplo, la tabla **t1** contiene las columnas **C1**, **C2**, **C3**, **C4** y **C5**. Para comprobar si las columnas **C2**, **C3** y **C4** se han actualizado (con un desencadenador UPDATE en la tabla **t1**), especifique un valor de 14. Para comprobar si sólo se ha actualizado la columna **C2**, especifique un valor de 2.

comparison_operator

Es el operador de comparación. Utilice el signo igual (=) para comprobar si todas las columnas especificadas en *updated_bitmask* se han actualizado. Utilice el símbolo mayor que (>) para comprobar si alguna de las columnas especificadas en *updated_bitmask* se han actualizado.

column_bitmask

Es la máscara de bits de enteros de las columnas que hay que comprobar para ver si se han actualizado o insertado.

Observaciones

A menudo se utilizan desencadenadores para exigir las reglas de empresa y la integridad de los datos. SQL Server proporciona integridad referencial declarativa (DRI, *Declarative Referential Integrity*) a través de las instrucciones de creación de tabla (ALTER TABLE y CREATE TABLE); sin embargo, DRI no proporciona integridad referencial entre bases de datos. Para exigir la integridad referencial (reglas acerca de la relación entre la clave principal y la clave externa de las tablas), utilice las restricciones de clave principal y externa (las palabras clave PRIMARY KEY y FOREIGN KEY de ALTER TABLE y CREATE TABLE). Si existen restricciones en la tabla de desencadenadores, se

comprueban después de la ejecución del desencadenador INSTEAD OF y antes de la de AFTER. Si no se respetan las restricciones, las acciones del desencadenador INSTEAD OF se deshacen y no se ejecuta (activa) el desencadenador AFTER.

El primer y último desencadenador AFTER que se ejecuta en una tabla se puede especificar mediante **sp_settriggerorder**. Sólo se puede especificar el primer y último desencadenador AFTER para cada una de las operaciones INSERT, UPDATE y DELETE de una tabla; si hay otros desencadenadores AFTER en la misma tabla, se ejecutan aleatoriamente.

Si una instrucción ALTER TRIGGER modifica el primer o último desencadenador, se elimina el primer o último atributo establecido en el desencadenador modificado, y el valor del orden se debe restablecer con **sp_settriggerorder**.

Un desencadenador AFTER se ejecuta sólo después de ejecutar correctamente la instrucción SQL desencadenadora, incluidas todas las acciones referenciales en cascada y las comprobaciones de restricciones asociadas con el objeto actualizado o eliminado. El desencadenador AFTER ve los efectos de la instrucción desencadenadora así como todas las acciones UPDATE y DELETE con referencias en cascada que causa la instrucción desencadenadora.

Limitaciones de los desencadenadores

CREATE TRIGGER debe ser la primera instrucción en el proceso por lotes y sólo se puede aplicar a una tabla.

Un desencadenador se crea solamente en la base de datos actual; sin embargo, un desencadenador puede hacer referencia a objetos que están fuera de la base de datos actual.

Si se especifica el nombre del propietario del desencadenador (para calificar el desencadenador), califique el nombre de la tabla de la misma forma.

La misma acción del desencadenador puede definirse para más de una acción del usuario (por ejemplo, INSERT y UPDATE) en la misma instrucción CREATE TRIGGER.

Los desencadenadores INSTEAD OF DELETE/UPDATE no pueden definirse en una tabla con una clave externa definida en cascada en la acción DELETE/UPDATE.

En un desencadenador se puede especificar cualquier instrucción SET. La opción SET elegida permanece en efecto durante la ejecución del desencadenador y, después, vuelve a su configuración anterior.

Cuando se activa un desencadenador, los resultados se devuelven a la aplicación que llama, exactamente igual que con los procedimientos almacenados. Para impedir que se devuelvan resultados a la aplicación debido a la activación de un desencadenador, no incluya las instrucciones SELECT que devuelven resultados ni las instrucciones que realizan una asignación variable en un desencadenador. Un desencadenador que incluya instrucciones

SELECT que devuelven resultados al usuario o instrucciones que realizan asignaciones de variables requiere un tratamiento especial; estos resultados devueltos tendrían que escribirse en cada aplicación en la que se permiten modificaciones a la tabla del desencadenador. Si es preciso que existan asignaciones de variable en un desencadenador, utilice una instrucción SET NOCOUNT al principio del mismo para eliminar la devolución de cualquier conjunto de resultados.

Un desencadenador DELETE no captura una instrucción TRUNCATE TABLE. Aunque una instrucción TRUNCATE TABLE es, de hecho, un desencadenador DELETE sin una cláusula WHERE (quita todas las filas), no se registra y, por tanto, no puede ejecutar un desencadenador. Dado que el permiso de la instrucción TRUNCATE TABLE es, de forma predeterminada, el del propietario de la tabla y no se puede transferir, sólo el propietario de la tabla debe preocuparse de invocar sin darse cuenta una instrucción TRUNCATE TABLE que no producirá la ejecución del desencadenador DELETE.

La instrucción WRITETEXT, ya se registre o no, no activa un desencadenador.

Las siguientes instrucciones Transact-SQL no están permitidas en un desencadenador:

ALTER DATABASE	CREATE DATABASE	DISK INIT
DISK RESIZE	DROP DATABASE	LOAD DATABASE
LOAD LOG	RECONFIGURE	RESTORE DATABASE
RESTORE LOG		

> **Nota** Debido a que SQL Server no admite desencadenadores definidos por el usuario en tablas del sistema, se recomienda que no se creen desencadenadores definidos por el usuario en tablas del sistema.

Desencadenadores múltiples

SQL Server permite que se creen varios desencadenadores por cada evento de modificación (DELETE, INSERT o UPDATE). Por ejemplo, si se ejecuta CREATE TRIGGER FOR UPDATE para una tabla que ya tiene un desencadenador UPDATE, se creará un desencadenador de actualización adicional. En las versiones anteriores, sólo se permitía un desencadenador por cada evento de modificación (INSERT, UPDATE, DELETE) en cada tabla.

> **Nota** El comportamiento predeterminado de CREATE TRIGGER (con un nivel de compatibilidad de 70) es agregar desencadenadores adicionales a los ya existentes si los nombres de desencadenadores son distintos. Si el nombre de los desencadenadores es el mismo, SQL Server devuelve un mensaje de error. Sin embargo, si el nivel de compatibilidad es igual o menor que 65, cualquier desencadenador creado con la

instrucción CREATE TRIGGER substituirá a los desencadenadores existentes del mismo tipo, incluso si los nombres de los desencadenadores son distintos.

Desencadenadores recursivos

SQL Server permite también la invocación recursiva de desencadenadores cuando el valor **recursive triggers** está habilitado en **sp_dboption**.

Los desencadenadores recursivos permiten dos tipos de repetición:

- Repetición indirecta
- Repetición directa

Con la repetición indirecta, una aplicación actualiza la tabla **T1**, que activa el desencadenador **TR1** para actualizar la tabla **T2**. En esta situación, el desencadenador **T2** activa y actualiza la tabla **T1**.

Con la repetición directa, la aplicación actualiza la tabla **T1**, que activa el desencadenador **TR1** para actualizar la tabla **T1**. Debido a que la tabla **T1** se ha actualizado, el desencadenador **TR1** se activa de nuevo, y así sucesivamente.

Este ejemplo utiliza ambas repeticiones de desencadenador, directa e indirecta. Suponga que en la tabla **T1** se han definido dos desencadenadores de actualización, **TR1** y **TR2**. El desencadenador **TR1** actualiza la tabla **T1** recursivamente. Una instrucción UPDATE ejecuta cada **TR1** y **TR2** una vez. Además, la ejecución de **TR1** desencadena la ejecución de **TR1** (recursivamente) y **TR2**. Las tablas **inserted** y **deleted** de un desencadenador dado contienen filas que corresponden sólo a la instrucción UPDATE que invocó al desencadenador.

Nota El comportamiento anterior sólo se produce si el valor **recursive triggers** de **sp_dboption** está habilitado. No hay un orden definido en el que se ejecuten los distintos desencadenadores definidos de un evento dado. Cada desencadenador debe ser independiente.

Deshabilitar **recursive triggers** sólo evita las repeticiones directas. Para deshabilitar también la repetición indirecta, establezca la opción de servidor **nested triggers** como 0 utilizando **sp_configure**.

Si alguno de los desencadenadores realiza una instrucción ROLLBACK TRANSACTION, no se ejecuta ningún desencadenador posterior, independientemente del nivel de anidamiento.

Desencadenadores anidados

Los desencadenadores pueden anidarse hasta un máximo de 32 niveles. Si un desencadenador cambia una tabla en la que hay otro desencadenador, el segundo se activa y puede, entonces, llamar a un tercero, y así sucesivamente. Si algún desencadenador de la cadena causa un bucle infinito, el nivel de anidamiento se habrá sobrepasado, con lo que se cancela el desencadenador. Para deshabilitar los desencadenadores anidados, establezca la opción **nested triggers** de **sp_configure** en 0 (desactivada). La configuración predeterminada permite desencadenadores anidados. Si los desencadenadores anidados están desactivados, los desencadenadores recursivos también se deshabilitan, independientemente del valor de **recursive triggers** de **sp_dboption**.

Resolución diferida de nombres

SQL Server permite que los procedimientos almacenados, desencadenadores y procesos por lotes de Transact-SQL hagan referencia a tablas que no existen en el momento de la compilación. Esta capacidad se denomina resolución diferida de nombres. Sin embargo, si los procedimientos almacenados, desencadenadores y procesos por lotes de Transact-SQL hacen referencia a una tabla definida en el procedimiento almacenado o desencadenador, se emitirá una advertencia en el momento de la creación sólo si el valor de nivel de compatibilidad (que se establece al ejecutar **sp_dbcmptlevel**) es igual a 65. Si se utiliza un proceso por lotes, la advertencia se emite en el momento de la compilación. Si la tabla a la que se hace referencia no existe, se devuelve un mensaje de error en tiempo de ejecución.

Para recuperar datos de una tabla o vista, un usuario debe tener permisos de la instrucción SELECT sobre la tabla o vista. Para actualizar el contenido de una tabla o vista, un usuario debe tener permisos de las instrucciones INSERT, DELETE y UPDATE sobre la tabla o vista.

Si existe un desencadenador INSTEAD OF en una vista, el usuario debe tener privilegios INSERT, DELETE y UPDATE sobre esa vista para ejecutar las instrucciones INSERT, DELETE y UPDATE en la vista, independientemente de si la ejecución realiza realmente esa operación en la vista.

Ejemplos

A. Utilizar un desencadenador con un mensaje de aviso

El siguiente desencadenador de ejemplo imprime un mensaje en el cliente cuando alguien intenta agregar o cambiar datos en la tabla **titles**.

```
USE pubs
IF EXISTS (SELECT name FROM sysobjects
      WHERE name = 'reminder' AND type = 'TR')
   DROP TRIGGER reminder
GO
CREATE TRIGGER reminder
ON titles
```

```
FOR INSERT, UPDATE
AS RAISERROR (50009, 16, 10)
GO
```

B. Utilizar un desencadenador con un mensaje de correo electrónico de aviso

Este ejemplo envía un mensaje de correo electrónico a una persona especificada (MaryM) cuando cambia la tabla **titles**.

```
USE pubs
IF EXISTS (SELECT name FROM sysobjects
      WHERE name = 'reminder' AND type = 'TR')
   DROP TRIGGER reminder
GO
CREATE TRIGGER reminder
ON titles
FOR INSERT, UPDATE, DELETE
AS
   EXEC master..xp_sendmail 'MaryM',
      'Don''t forget to print a report for the distributors.'
GO
```

C. Utilizar una regla de empresa desencadenador entre las tablas employee y jobs

Debido a que las restricciones CHECK sólo pueden hacer referencia a las columnas en que se han definido las restricciones de columna o de tabla, cualquier restricción de referencias cruzadas, en este caso, reglas de empresa, debe definirse como desencadenadores.

Este ejemplo crea un desencadenador que, cuando se inserta o se cambia un nivel de trabajo de empleado, comprueba que el nivel especificado del trabajo del empleado (**job_lvls**) en el que se basan los salarios se encuentra en el intervalo definido para el trabajo. Para obtener el intervalo adecuado, debe hacerse referencia a la tabla **jobs**.

```
USE pubs
IF EXISTS (SELECT name FROM sysobjects
      WHERE name = 'employee_insupd' AND type = 'TR')
   DROP TRIGGER employee_insupd
GO
CREATE TRIGGER employee_insupd
ON employee
FOR INSERT, UPDATE
AS
/* Get the range of level for this job type from the jobs table. */
DECLARE @min_lvl tinyint,
   @max_lvl tinyint,
   @emp_lvl tinyint,
   @job_id smallint
SELECT @min_lvl = min_lvl,
   @max_lvl = max_lvl,
   @emp_lvl = i.job_lvl,
   @job_id = i.job_id
FROM employee e INNER JOIN inserted i ON e.emp_id = i.emp_id
   JOIN jobs j ON j.job_id = i.job_id
```

```
IF (@job_id = 1) and (@emp_lvl <> 10)
BEGIN
   RAISERROR ('Job id 1 expects the default level of 10.', 16, 1)
   ROLLBACK TRANSACTION
END
ELSE
IF NOT (@emp_lvl BETWEEN @min_lvl AND @max_lvl)
BEGIN
   RAISERROR ('The level for job_id:%d should be between %d and %d.',
      16, 1, @job_id, @min_lvl, @max_lvl)
   ROLLBACK TRANSACTION
END
```

D. Utilizar la resolución diferida de nombres

El ejemplo siguiente crea dos desencadenadores para ilustrar la resolución diferida de nombres.

```
USE pubs
IF EXISTS (SELECT name FROM sysobjects
      WHERE name = 'trig1' AND type = 'TR')
   DROP TRIGGER trig1
GO
-- Creating a trigger on a nonexistent table.
CREATE TRIGGER trig1
on authors
FOR INSERT, UPDATE, DELETE
AS
   SELECT a.au_lname, a.au_fname, x.info
   FROM authors a INNER JOIN does_not_exist x
      ON a.au_id = x.au_id
GO
-- Here is the statement to actually see the text of the trigger.
SELECT o.id, c.text
FROM sysobjects o INNER JOIN syscomments c
   ON o.id = c.id
WHERE o.type = 'TR' and o.name = 'trig1'

-- Creating a trigger on an existing table, but with a nonexistent
-- column.
USE pubs
IF EXISTS (SELECT name FROM sysobjects
      WHERE name = 'trig2' AND type = 'TR')
   DROP TRIGGER trig2
GO
CREATE TRIGGER trig2
ON authors
FOR INSERT, UPDATE
AS
   DECLARE @fax varchar(12)
   SELECT @fax = phone
   FROM authors
GO
-- Here is the statement to actually see the text of the trigger.
SELECT o.id, c.text
FROM sysobjects o INNER JOIN syscomments c
```

```
   ON o.id = c.id
WHERE o.type = 'TR' and o.name = 'trig2'
```

E. Utilizar COLUMNS_UPDATED

En este ejemplo se crean dos tablas: una tabla **employeeData** y una tabla **auditEmployeeData**. La tabla **employeeData**, que contiene información confidencial de los sueldos de los empleados, puede ser modificada por los miembros del departamento de recursos humanos. Si se cambia el número de seguridad social del empleado, el sueldo anual o el número de cuenta bancaria, se genera un registro de auditoría y se inserta en la tabla de auditoría **auditEmployeeData**.

Con la función COLUMNS_UPDATED(), es posible comprobar rápidamente cualquier cambio en estas columnas que contienen información confidencial de los empleados. COLUMNS_UPDATED() sólo se puede utilizar de esta manera para intentar detectar modificaciones en las primeras 8 columnas de la tabla.

```
USE pubs
IF EXISTS(SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
   WHERE TABLE_NAME = 'employeeData')
   DROP TABLE employeeData
IF EXISTS(SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
   WHERE TABLE_NAME = 'auditEmployeeData')
   DROP TABLE auditEmployeeData
GO
CREATE TABLE employeeData (
   emp_id int NOT NULL,
   emp_bankAccountNumber char (10) NOT NULL,
   emp_salary int NOT NULL,
   emp_SSN char (11) NOT NULL,
   emp_lname nchar (32) NOT NULL,
   emp_fname nchar (32) NOT NULL,
   emp_manager int NOT NULL
   )
GO
CREATE TABLE auditEmployeeData (
   audit_log_id uniqueidentifier DEFAULT NEWID(),
   audit_log_type char (3) NOT NULL,
   audit_emp_id int NOT NULL,
   audit_emp_bankAccountNumber char (10) NULL,
   audit_emp_salary int NULL,
   audit_emp_SSN char (11) NULL,
   audit_user sysname DEFAULT SUSER_SNAME(),
   audit_changed datetime DEFAULT GETDATE()
   )
GO
CREATE TRIGGER updEmployeeData
ON employeeData
FOR update AS
/*Check whether columns 2, 3 or 4 has been updated. If any or all of
columns 2, 3 or 4 have been changed, create an audit record. The bitmask
is: power(2,(2-1))+power(2,(3-1))+power(2,(4-1)) = 14. To check if all
columns 2, 3, and 4 are updated, use = 14 in place of >0 (below).*/
```

```
   IF (COLUMNS_UPDATED() & 14) > 0
/*Use IF (COLUMNS_UPDATED() & 14) = 14 to see if all of columns 2, 3, and
4 are updated.*/
      BEGIN
-- Audit OLD record.
      INSERT INTO auditEmployeeData
         (audit_log_type,
         audit_emp_id,
         audit_emp_bankAccountNumber,
         audit_emp_salary,
         audit_emp_SSN)
         SELECT 'OLD',
            del.emp_id,
            del.emp_bankAccountNumber,
            del.emp_salary,
            del.emp_SSN
         FROM deleted del

-- Audit NEW record.
      INSERT INTO auditEmployeeData
         (audit_log_type,
         audit_emp_id,
         audit_emp_bankAccountNumber,
         audit_emp_salary,
         audit_emp_SSN)
         SELECT 'NEW',
            ins.emp_id,
            ins.emp_bankAccountNumber,
            ins.emp_salary,
            ins.emp_SSN
         FROM inserted ins
   END
GO

/*Inserting a new employee does not cause the UPDATE trigger to fire.*/
INSERT INTO employeeData
   VALUES ( 101, 'USA-987-01', 23000, 'R-M53550M', N'Mendel', N'Roland',
32)
GO

/*Updating the employee record for employee number 101 to change the
salary to 51000 causes the UPDATE trigger to fire and an audit trail to
be produced.*/

UPDATE employeeData
   SET emp_salary = 51000
   WHERE emp_id = 101
GO
SELECT * FROM auditEmployeeData
GO

/*Updating the employee record for employee number 101 to change both the
bank account number and social security number (SSN) causes the UPDATE
trigger to fire and an audit trail to be produced.*/

UPDATE employeeData
   SET emp_bankAccountNumber = '133146A0', emp_SSN = 'R-M53550M'
```

```
   WHERE emp_id = 101
GO
SELECT * FROM auditEmployeeData
GO
```

F. Utilizar COLUMNS_UPDATED para probar más de 8 columnas

Si tiene que probar actualizaciones que afectan a otras columnas que no sean las 8 primeras de una tabla, debe utilizar la función SUBSTRING para probar si COLUMNS_UPDATED devuelve el bit correcto. Este ejemplo prueba las actualizaciones que afectan a las columnas 3, 5 o 9 de la tabla **Northwind.dbo.Customers**.

```
USE Northwind
DROP TRIGGER  tr1
GO
CREATE TRIGGER tr1 ON Customers
FOR UPDATE AS
   IF ( (SUBSTRING(COLUMNS_UPDATED(),1,1)=power(2,(3-1))
      + power(2,(5-1)))
      AND (SUBSTRING(COLUMNS_UPDATED(),2,1)=power(2,(1-1)))
      )
   PRINT 'Columns 3, 5 and 9 updated'
GO

UPDATE Customers
   SET ContactName=ContactName,
      Address=Address,
      Country=Country
GO
```

ALTER TRIGGER

Altera la definición de un desencadenador creado previamente por la instrucción CREATE TRIGGER.

Sintaxis

ALTER TRIGGER *trigger_name*
ON **(** *table* | view **)**
[WITH ENCRYPTION]
{
  { **(** FOR | AFTER | INSTEAD OF **)** { [DELETE] [**,**] [INSERT] [**,**] [UPDATE] }
    [NOT FOR REPLICATION]
    AS
    sql_statement [*...n*]
  }
  |
  { **(** FOR | AFTER | INSTEAD OF **)** { [INSERT] [**,**] [UPDATE] }

```
    [ NOT FOR REPLICATION ]
    AS
    { IF UPDATE ( column )
    [ { AND | OR } UPDATE ( column ) ]
    [ ...n ]
    | IF ( COLUMNS_UPDATED ( ) { bitwise_operator } updated_bitmask )
    { comparison_operator } column_bitmask [ ...n ]
    }
    sql_statement [ ...n ]
  }
}
```

Argumentos

trigger_name

Es el desencadenador existente que se va a alterar.

table | *view*

Es la tabla o la vista en que se ejecuta el desencadenador.

WITH ENCRYPTION

Cifra las entradas **syscomments** que contienen el texto de la instrucción ALTER TRIGGER. El uso de WITH ENCRYPTION impide que el desencadenador se publique como parte de la duplicación de SQL Server.

> **Nota** Si se creó una definición anterior de desencadenador mediante WITH ENCRYPTION o RECOMPILE, estas opciones sólo estarán habilitadas si se incluyen en ALTER TRIGGER.

AFTER

Especifica que el desencadenador se activa sólo después de que se ejecute correctamente la instrucción SQL desencadenadora. Todas las acciones referenciales en cascada y las comprobaciones de restricciones también deben haber sido correctas antes de ejecutar este desencadenador.

AFTER es el valor predeterminado, si se especifica sólo la palabra clave FOR.

Los desencadenadores AFTER sólo pueden definirse en tablas.

INSTEAD OF

Especifica que se ejecuta el desencadenador en vez de la instrucción SQL desencadenadora, por lo que se suplantan las acciones de las instrucciones desencadenadoras.

Como máximo, se puede definir un desencadenador INSTEAD OF por cada instrucción INSERT, UPDATE o DELETE en cada tabla o vista. No obstante, en las vistas es posible definir otras vistas que tengan su propio desencadenador INSTEAD OF.

No se permiten desencadenadores INSTEAD OF en vistas creadas con WITH CHECK OPTION. SQL Server generará un error si se agrega un desencadenador INSTEAD OF a una vista para la que se haya especificado WITH CHECK OPTION. El usuario debe quitar esta opción mediante ALTER VIEW antes de definir el desencadenador INSTEAD OF.

{ [DELETE] [**,**] [INSERT] [**,**] [UPDATE] } | { [INSERT] [**,**] [UPDATE]}

Son palabras clave que especifican qué instrucciones de modificación de datos activan el desencadenador cuando se intentan en esta tabla o vista. Se debe especificar al menos una opción. En la definición del desencadenador se permite cualquier combinación de éstas, en cualquier orden. Si especifica más de una opción, sepárelas con comas.

Para los desencadenadores INSTEAD OF, no se permite la opción DELETE en tablas que tengan una relación de integridad referencial que especifica una acción ON DELETE en cascada. Igualmente, no se permite la opción UPDATE en tablas que tengan una relación de integridad referencial que especifica una acción ON UPDATE en cascada.

NOT FOR REPLICATION

Indica que el desencadenador no debe ejecutarse cuando un inicio de sesión de duplicación, como **sqlrepl**, modifica la tabla involucrada en el desencadenador.

AS

Son las acciones que va a llevar a cabo el desencadenador.

sql_statement

Son las condiciones y acciones del desencadenador.

n

Se trata de un marcador de posición que indica que se pueden incluir varias instrucciones Transact-SQL en el desencadenador.

IF UPDATE **(***column***)**

Prueba una acción INSERT o UPDATE en una columna especificada y no se utiliza con operaciones DELETE.

UPDATE(*column*) puede utilizarse en cualquier parte dentro del cuerpo del desencadenador.

{AND | OR}

Especifica otra columna para probar una acción INSERT o UPDATE.

column

Es el nombre de la columna que se va a probar para una acción INSERT o UPDATE.

IF (COLUMNS_UPDATED())

Prueba, sólo en un desencadenador INSERT o UPDATE, si la columna o columnas mencionadas se insertaron o se actualizaron. COLUMNS_UPDATED devuelve un patrón de bits **varbinary** que indica qué columnas de la tabla se insertaron o se actualizaron.

COLUMNS_UPDATED puede utilizarse en cualquier parte dentro del cuerpo del desencadenador.

bitwise_operator

Es el operador de bits que se utilizará en las comparaciones.

updated_bitmask

Es la máscara de bits de enteros de las columnas realmente actualizadas o insertadas. Por ejemplo, la tabla **t1** contiene las columnas **C1**, **C2**, **C3**, **C4** y **C5**. Para comprobar si las columnas **C2**, **C3** y **C4** se han actualizado (con un desencadenador UPDATE en la tabla **t1**), especifique el valor **14**. Para comprobar si sólo se ha actualizado la columna **C2**, especifique el valor **2**.

comparison_operator

Es el operador de comparación. Utilice el signo igual (=) para comprobar si todas las columnas especificadas en *updated_bitmask* se han actualizado. Utilice el símbolo mayor que (>) para comprobar si ninguna o algunas de las columnas especificadas en *updated_bitmask* se han actualizado.

column_bitmask

Es la máscara de bits de enteros de la columna que se va a comprobar.

ALTER TRIGGER acepta vistas actualizables manualmente mediante desencadenadores INSTEAD OF en tablas y vistas. Microsoft® SQL Server™ aplica ALTER TRIGGER de la misma forma a todos los tipos de desencadenadores (AFTER, INSTEAD-OF).

El primer y último desencadenador AFTER que se ejecuta en una tabla se puede especificar mediante **sp_settriggerorder**. Sólo se pueden especificar un primer y un último desencadenador AFTER en una tabla; si hay otros desencadenadores AFTER en la misma tabla, se ejecutarán en una secuencia no definida.

Si una instrucción ALTER TRIGGER modifica el primer o último desencadenador, se elimina el primer o último atributo establecido en el desencadenador modificado, y el valor del orden se debe restablecer con **sp_settriggerorder**.

Un desencadenador AFTER se ejecuta sólo después de ejecutar correctamente la instrucción SQL desencadenadora, incluidas todas las acciones referenciales en cascada y las comprobaciones de restricciones asociadas con el objeto actualizado o eliminado. La operación del desencadenador AFTER comprueba los efectos de la instrucción desencadenadora y todas las acciones UPDATE y DELETE referenciales y en cascada generadas por la instrucción desencadenadora.

Cuando una acción DELETE en una tabla de referencia o secundaria es el resultado de una instrucción DELETE en CASCADE de la tabla principal y el desencadenador INSTEAD OF de DELETE está definido en esta tabla secundaria, el desencadenador se pasa por alto y se ejecuta la acción DELETE.

Ejemplos

El ejemplo siguiente crea un desencadenador que imprime para el cliente un mensaje definido por el usuario cuando un usuario intenta agregar o cambiar datos en la tabla **roysched**. A continuación, el desencadenador se altera mediante ALTER TRIGGER para aplicarlo sólo en las actividades INSERT. Este desencadenador es útil porque recuerda al usuario que actualiza o inserta filas en la tabla que notifique los cambios también a los autores y editores de los libros.

```
USE pubs
GO
CREATE TRIGGER royalty_reminder
ON roysched
WITH ENCRYPTION
FOR INSERT, UPDATE
AS RAISERROR (50009, 16, 10)

-- Now, alter the trigger.
USE pubs
GO
ALTER TRIGGER royalty_reminder
ON roysched
FOR INSERT
AS RAISERROR (50009, 16, 10)
```

DROP TRIGGER

Quita uno o más desencadenadores de la base de datos actual.

Sintaxis

DROP TRIGGER { *trigger* } [**,***...n*]

Argumentos

trigger

Es el nombre del desencadenador que se va a quitar. Los nombres de desencadenadores deben seguir las reglas de los identificadores.

n

Es un marcador de posición que indica que se pueden especificar varios desencadenadores.

Observaciones

Puede eliminar un desencadenador si quita éste o quita la tabla del desencadenador. Cuando se quita una tabla, también se quitan todos los desencadenadores asociados. Cuando se quita un desencadenador, se quita la información acerca del desencadenador de las tablas de sistema **sysobjects** y **syscomments**.

Utilice DROP TRIGGER y CREATE TRIGGER para cambiar el nombre de un desencadenador. Utilice ALTER TRIGGER para cambiar la definición de un desencadenador.

Ejemplos

Este ejemplo quita el desencadenador **employee_insupd**.

```
USE pubs
IF EXISTS (SELECT name FROM sysobjects
      WHERE name = 'employee_insupd' AND type = 'TR')
   DROP TRIGGER employee_insupd
GO
```

VISTAS

CREATE VIEW

Crea una tabla virtual que representa los datos de una o más tablas de una forma alternativa. CREATE VIEW debe ser la primera instrucción en una secuencia de consultas.

Sintaxis

CREATE VIEW [< *database_name* > **.**] [< *owner* > **.**] *view_name* [**(** *column* [**,***...n*] **)**]
[WITH < view_attribute > [**,**...n]]
AS
select_statement
[WITH CHECK OPTION]

< view_attribute > ::=
{ ENCRYPTION | SCHEMABINDING | VIEW_METADATA }

Argumentos

view_name

Es el nombre de la vista. Los nombres de las vistas deben cumplir las reglas de los identificadores. Especificar el propietario de la vista es opcional.

column

Es el nombre que se va a utilizar para una columna en una vista. Sólo es necesario asignar un nombre a una columna en CREATE VIEW cuando una columna proviene de una expresión aritmética, una función o una constante; cuando dos o más columnas puedan tener el mismo nombre (normalmente, debido a una combinación); o cuando una columna de una vista recibe un nombre distinto al de la columna de la que proviene. Los nombres de columna se pueden asignar también en la instrucción SELECT.

Si no se especifica *column*, las columnas de la vista adquieren los mismos nombres que las columnas de la instrucción SELECT.

> **Nota** En las columnas de la vista, los permisos de un nombre de columna se aplican a través de una instrucción CREATE VIEW o ALTER VIEW, independientemente del origen de los datos subyacentes. Por ejemplo, si se conceden permisos sobre la columna **title_id** de una instrucción CREATE VIEW, una instrucción ALTER VIEW puede llamar a la columna **title_id** con un nombre de columna distinto, por ejemplo **qty**, y seguir teniendo los permisos asociados con la vista que utiliza **title_id**.

n

Es un marcador de posición que indica que se pueden especificar varias columnas.

AS

Son las acciones que va a llevar a cabo la vista.

select_statement

Es la instrucción SELECT que define la vista. Puede utilizar más de una tabla y otras vistas. Para seleccionar los objetos a los que se hace referencia en la cláusula SELECT de una vista creada, es necesario tener los permisos adecuados.

Una vista no tiene por qué ser un simple subconjunto de filas y de columnas de una tabla determinada. Una vista se puede crear con más de una tabla o con otras vistas, mediante una cláusula SELECT de cualquier complejidad.

En una definición de vista indizada, la instrucción SELECT debe ser una instrucción para una única tabla o una instrucción JOIN con agregaciones opcionales.

Hay unas cuantas restricciones en las cláusulas SELECT en una definición de vista. Una instrucción CREATE VIEW no puede:

- Incluir las cláusulas COMPUTE o COMPUTE BY.
- Incluir la cláusula ORDER BY, a menos que también haya una cláusula TOP en la lista de selección de la instrucción SELECT.
- Incluir la palabra clave INTO.
- Hacer referencia a una tabla temporal o a una variable de tabla.

Como *select_statement* utiliza la instrucción SELECT, es válido utilizar las sugerencias <join_hint> y <table_hint> como se especifican en la cláusula FROM. Se pueden utilizar funciones en *select_statement*.

select_statement puede utilizar varias instrucciones SELECT separadas con UNION o UNION ALL.

WITH CHECK OPTION

Exige que todas las instrucciones de modificación de datos ejecutadas contra la vista se adhieran a los criterios establecidos en *select_statement*. Cuando una fila se modifica mediante una vista, WITH CHECK OPTION garantiza que los datos permanecerán visibles en toda la vista después de confirmar la modificación.

WITH ENCRYPTION

Indica que SQL Server cifra las columnas de la tabla del sistema que contienen el texto de la instrucción CREATE VIEW. Utilizar WITH ENCRYPTION evita que la vista se publique como parte de la duplicación de SQL Server.

SCHEMABINDING

Enlaza la vista al esquema. Cuando se especifica SCHEMABINDING, *select_statement* debe incluir los nombres con dos partes (propietario.objeto) de las tablas, vistas o funciones definidas por el usuario a las que se hace referencia.

Las vistas o las tablas que participan en una vista creada con la cláusula de enlace de esquema no se pueden quitar ni alterar, de forma que deja de tener un enlace de esquema. De lo contrario, SQL Server genera un error. Además, las instrucciones ALTER TABLE sobre tablas que participan en vistas que tienen enlaces de esquemas provocarán un error si estas instrucciones afectan a la definición de la vista.

VIEW_METADATA

Especifica que SQL Server devolverá a las API de DBLIB, ODBC y OLE DB la información de metadatos sobre la vista, en vez de las tablas o tabla base, cuando se soliciten los metadatos del modo de exploración para una consulta que hace referencia a la vista. Los metadatos del modo de exploración son metadatos adicionales devueltos por SQL Server a las API DB-LIB, ODBC y OLE DB del cliente, que permiten a las API del cliente implementar cursores actualizables en el cliente. Los metadatos del modo de exploración incluyen información sobre la tabla base a la que pertenecen las columnas del conjunto de resultados.

Para las vistas creadas con la opción VIEW_METADATA, los metadatos del modo de exploración devuelven el nombre de vista en vez de los nombres de la tabla base cuando se describen las columnas de la vista en el conjunto de resultados.

Cuando se crea una vista WITH VIEW_METADATA, todas sus columnas (excepto **timestamp**) son actualizables si la vista tiene los desencadenadores INSERT o UPDATE INSTEAD OF. Consulte Vistas actualizables, más adelante en este capítulo.

Observaciones

Una vista sólo se puede crear en la base de datos actual. Una vista puede hacer referencia a un máximo de 1.024 columnas.

Cuando se realiza una consulta a través de una vista, Microsoft® SQL Server™ comprueba que todos los objetos de base de datos a los que se hace referencia en la instrucción existen, que son válidos en el contexto de la instrucción y que las instrucciones de modificación de datos no infringen ninguna regla de integridad de los datos. Las comprobaciones que no son

correctas devuelven un mensaje de error. Las comprobaciones correctas convierten la acción en una acción contra las tablas subyacentes.

Si una vista depende de una tabla (o vista) que se ha eliminado, SQL Server produce un mensaje de error si alguien trata de utilizar la vista. Si se crea una nueva tabla (o vista), y la estructura de la tabla no cambia de la tabla base anterior para substituir a la eliminada, se puede volver a utilizar la vista de nuevo. Si cambia la estructura de la nueva tabla (o vista), es necesario eliminar la vista y volver a crearla.

Cuando se crea una vista, el nombre de la vista se almacena en la tabla **sysobjects**. La información acerca de las columnas definidas en una vista se agrega a la tabla **syscolumns** y la información acerca de las dependencias de la vista se agrega a la tabla **sysdepends**. Además, el texto de la instrucción CREATE VIEW se agrega a la tabla **syscomments**. Esto es similar a un procedimiento almacenado; cuando la vista se ejecuta por primera vez, sólo su árbol de consulta se almacena en la caché de procedimientos. Cada vez que se tiene acceso a una vista, su plan de ejecución se vuelve a compilar.

El resultado de una consulta que utiliza un índice de una vista definido con expresiones **numeric** o **float** podría diferir de una consulta similar que no utiliza el índice de la vista. Esta diferencia se podría deber a errores de redondeo durante las acciones INSERT, DELETE o UPDATE en las tablas subyacentes.

Cuando se crea una vista, SQL Server guarda la configuración de SET QUOTED_IDENTIFIER y SET ANSI_NULLS. Estos valores originales se restauran cuando se utiliza la vista. Por tanto, cualquier configuración de sesión de cliente de SET QUOTED_IDENTIFIER y SET ANSI_NULLS se omite al obtener acceso a la vista.

> **Nota** El que SQL Server interprete una cadena vacía como un espacio simple o como una verdadera cadena vacía se controla mediante el valor de **sp_dbcmptlevel**. Si el nivel de compatibilidad es menor o igual que 65, SQL Server interpreta las cadenas vacías como espacios simples. Si el nivel de compatibilidad es mayor o igual que 70, SQL Server interpreta las cadenas vacías como espacios vacíos.

Vistas actualizables

Microsoft SQL Server 2000 mejora la clase de vistas actualizables de dos maneras:

- **Desencadenadores INSTEAD OF:** se pueden crear desencadenadores INSTEAD OF en una vista para que sea actualizable. El desencadenador INSTEAD OF se ejecuta en lugar de la instrucción de modificación de datos donde se define el desencadenador. Este desencadenador permite al usuario especificar el conjunto de acciones que hay que realizar para procesar la instrucción de modificación de datos. Por lo tanto, si existe un desencadenador INSTEAD OF para una vista en una instrucción de modificación de datos determinada (INSERT, UPDATE o DELETE), la vista correspondiente se puede actualizar mediante esa instrucción.

- **Vistas con particiones:** Si la vista es del tipo denominado 'vista con particiones', se puede actualizar con determinadas restricciones. Las vistas con particiones y su actualización se tratarán más adelante en este tema.

  Si es necesario, SQL Server puede distinguir **Local Partitioned Views** como las vistas en las que todas las tablas participantes y la vista se encuentran en el mismo SQL Server, y **Distributed Partitioned Views** como las vistas en las que al menos una de las tablas de la vista reside en otro servidor (remoto).

Si una vista no tiene desencadenadores INSTEAD OF, o si no es una vista con particiones, sólo se puede actualizar si se cumplen las siguientes condiciones:

- El argumento *select_statement* no tiene funciones de agregado en la lista de selección y no contiene las cláusulas TOP, GROUP BY, UNION (a menos que la vista tenga particiones, como se describe más adelante en este capítulo) o DISTINCT. Las funciones de agregado se pueden utilizar en una subconsulta en la cláusula FROM, siempre y cuando los valores devueltos por las funciones no se modifiquen.

- El argumento *instrucciónSelección* no tiene columnas derivadas en la lista de selección. Las columnas derivadas son columnas de conjunto de resultados formadas por cualquier elemento que no sea una expresión de columna simple, tal como el uso de funciones u operadores de adición y sustracción.

- La cláusula FROM de *select_statement* hace referencia al menos a una tabla. *select_statement* debe tener más de una expresión no tabular, es decir, expresiones que no provienen de una tabla. Por ejemplo, esta vista no se puede actualizar:

```
CREATE VIEW NoTable AS
SELECT GETDATE() AS CurrentDate,
       @@LANGUAGE AS CurrentLanguage,
       CURRENT_USER AS CurrentUser
```

Las instrucciones INSERT, UPDATE y DELETE también deben cumplir determinados requisitos para poder hacer referencia a una vista actualizable, como se indica en las condiciones anteriores. Las instrucciones UPDATE e INSERT sólo pueden hacer referencia a una vista si ésta es actualizable y la instrucción UPDATE o INSERT está escrita de forma que sólo modifique los datos de una de las tablas base a la que se hace referencia en la cláusula FROM de la vista. Una instrucción DELETE sólo puede hacer referencia a una vista actualizable si la vista hace referencia exactamente a una tabla en su cláusula FROM.

Vistas con particiones

Una vista con particiones es una vista que se define mediante la instrucción UNION ALL, es decir, uniendo todas las tablas miembro estructuradas de la misma manera pero almacenadas en diferentes tablas del mismo servidor SQL Server o en un grupo de

servidores SQL Server 2000 autónomos, denominados Servidores federados de SQL Server 2000.

Por ejemplo, si tiene los datos de la tabla **Customers** distribuidos en tres tablas miembro situadas en tres ubicaciones de servidor (**Customers_33** en **Server1**, **Customers_66** en **Server2** y **Customers_99** en **Server3**), una vista con particiones en **Server1** se definiría de esta manera:

```
--Partitioned view as defined on Server1
CREATE VIEW Customers
AS
--Select from local member table
SELECT *
FROM CompanyData.dbo.Customers_33
UNION ALL
--Select from member table on Server2
SELECT *
FROM Server2.CompanyData.dbo.Customers_66
UNION ALL
--Select from mmeber table on Server3
SELECT *
FROM Server3.CompanyData.dbo.Customers_99
```

En general, se dice que una vista tiene particiones si tiene la siguiente forma:

```
SELECT <select_list1>
FROM T1
UNION ALL
SELECT <select_list2>
FROM T2
UNION ALL
...
SELECT <select_list n>
FROM T n
```

Condiciones para la creación de vistas con particiones

1. Lista SELECT
    - Todas las columnas de las tablas miembro deben seleccionarse en la lista de columnas de la definición de la vista.

    - Las columnas de la misma posición ordinal de cada lista SELECT deben ser del mismo tipo, incluidas las intercalaciones. No es suficiente que las columnas sean de tipos implícitamente convertibles, como sucede normalmente con UNION.

      Además, ***al menos una*** columna, por ejemplo, <col>, debe aparecer en todas las listas SELECT en la misma posición ordinal. Esta columna <col> debe definirse de manera que las tablas miembro T1, ..., T*n* tengan restricciones CHECK C1, ..., C*n* definidas en <col> respectivamente.

La restricción C1 definida en la tabla T1 debe tener esta forma:

```
C1 ::= < simple_interval > [ OR < simple_interval > OR ...]
< simple_interval > :: =
    < col > { < | > | <= | >= | = }
    | < col > BETWEEN < value1 > AND < value2 >
    | < col > IN ( value_list )
    | < col > { > | >= } < value1 > AND
        < col > { < | <= } < value2 >
```

- Las restricciones deben estar definidas de manera que cualquier valor de <col> pueda satisfacer ***al menos una*** de las restricciones C1, ..., C*n* para que las restricciones formen un conjunto de intervalos no combinados o que no se solapen. La columna <col> en la que se definen las restricciones no combinadas se denomina 'columna de partición'. Observe que la columna de partición puede tener diferentes nombres en las tablas subyacentes. Las restricciones deben tener un estado **enabled**, es decir, estar habilitadas, para que puedan cumplir las condiciones de la columna de partición. Si las restricciones están deshabilitadas, habilite de nuevo la comprobación de restricciones con las opciones WITH CHECK o CHECK *constraint_name* de ALTER TABLE.

  Éstos son algunos ejemplos de conjuntos de restricciones válidos:

```
{ [col < 10], [col between 11 and 20] , [col > 20] }
{ [col between 11 and 20], [col between 21 and 30], [col
between 31 and 100] }
```

- No se puede utilizar la misma columna varias veces en la lista SELECT.

2. Columna de partición
   - La columna de partición forma parte de la restricción PRIMARY KEY de la tabla.
   - No puede ser una columna calculada.
   - Si existe más de una restricción en la misma columna de una tabla miembro, SQL Server ignora todas las restricciones y no las tendrá en cuenta al determinar si la vista es con particiones o no lo es. Para cumplir las condiciones de la vista con particiones, solamente debe existir una restricción de partición en la columna de partición.
3. Tablas miembro, o tablas subyacentes T1, ..., T*n*
   - Las tablas pueden ser locales o tablas de otros servidores SQL Server a los que se hace referencia mediante un nombre de cuatro partes o un nombre basado en OPENDATASOURCE u OPENROWSET. (La sintaxis de OPENDATASOURCE y OPENROWSET puede especificar un nombre de tabla, pero no una consulta de paso a través.)

Si algunas tablas miembro son remotas, la vista se denomina ***vista con particiones distribuida***, y se aplican algunas condiciones adicionales. Se tratarán más adelante en esta sección.

- La misma tabla no puede aparecer dos veces en el conjunto de tablas que se está combinando con la instrucción UNION ALL.
- Las tablas miembro no pueden tener índices creados en columnas calculadas de la tabla.
- Las tablas miembro deben tener todas las restricciones PRIMARY KEY en un número de columnas idéntico.
- Todas las tablas miembro de la vista deben incluir la misma configuración de valores de relleno ANSI (se configura con la opción **user options** de **sp_configure** o con la opción SET).

Condiciones para modificar vistas con particiones

Solamente las versiones Developer y Enterprise de SQL Server 2000 admiten las operaciones INSERT, UPDATE y DELETE en las vistas con particiones. Para modificar las vistas con particiones, las instrucciones deben cumplir estas condiciones:

- La instrucción INSERT debe proporcionar valores para todas las columnas de la vista, incluso si las tablas miembro subyacentes tienen restricciones DEFAULT en dichas columnas o admiten valores NULL. En las columnas de tabla miembro con definiciones DEFAULT, las instrucciones no pueden usar explícitamente la palabra clave DEFAULT.
- El valor que se va a insertar en la columna de partición debe cumplir al menos una de las restricciones subyacentes; en caso contrario, se infringirá la restricción y la acción INSERT causará un error.
- Las instrucciones UPDATE no pueden especificar la palabra clave DEFAULT como un valor de la cláusula SET, incluso si la columna tiene un valor DEFAULT definido en la tabla miembro correspondiente.
- Las columnas PRIMARY KEY no pueden modificarse en una instrucción UPDATE si las tablas miembro tienen columnas de tipo **text**, **ntext**, o **image**.
- Las columnas de la vista que sean columnas IDENTITY en una o varias tablas miembro no se pueden modificar mediante una instrucción INSERT o UPDATE.
- Si una de las tablas miembro contiene una columna **timestamp**, la vista no se puede modificar mediante una instrucción INSERT o UPDATE.

- Las acciones INSERT, UPDATE y DELETE no se permiten en vistas con particiones si hay una autocombinación con la vista o con cualquiera de las tablas miembro de la instrucción.

> **Nota** Para actualizar una vista con particiones, el usuario debe tener permisos INSERT, UPDATE y DELETE sobre las tablas miembro.

Condiciones adicionales para las vistas con particiones distribuidas

Para vistas con particiones distribuidas (cuando una o varias tablas miembro son remotas), se aplican las siguientes condiciones adicionales:

- Se iniciará una transacción distribuida para garantizar la atomicidad en todos los nodos a los que afecta la actualización.

- La opción XACT_ABORT SET debe ser ON para que las instrucciones INSERT, UPDATE o DELETE funcionen.

- Cualquier columna **smallmoney** y **smalldatetime** de las tablas remotas a las que se hace referencia en una vista con particiones se asigna como **money** y **datetime,** respectivamente. Por lo tanto, las columnas correspondientes (en la misma posición ordinal de la lista select) de las tablas locales deben ser **money** y **datetime**.

- Ningún servidor vinculado de la vista con particiones puede ser un servidor vinculado en bucle de retorno (un servidor vinculado que apunta al mismo SQL Server).

El valor de la opción SET ROWCOUNT se pasa por alto para las acciones INSERT, UPDATE y DELETE que implican vistas con particiones y tablas remotas actualizables.

Cuando las tablas miembro y la definición de la vista con particiones están en su lugar, Microsoft SQL Server 2000 crea planes inteligentes que utilizan las consultas de forma eficaz para tener acceso a los datos de las tablas miembro. Con las definiciones de la restricción CHECK, el procesador de consultas asigna la distribución de valores clave entre las tablas miembro. Cuando un usuario ejecuta una consulta, el procesador de la consulta compara la asignación con los valores especificados en la cláusula WHERE, y crea un plan de ejecución con una transferencia mínima de datos entre los servidores de miembros. Por lo tanto, aunque algunas tablas miembro estén ubicadas en servidores remotos, SQL Server 2000 resolverá las consultas distribuidas de manera que la cantidad de datos distribuidos que hay que transferir será mínima.

Consideraciones acerca de la duplicación

Para crear vistas con particiones en tablas miembro implicadas en la duplicación, deben tenerse en cuenta las consideraciones siguientes:

- Si las tablas subyacentes intervienen en la duplicación de mezcla o en la duplicación transaccional con suscriptores de actualización, la columna uniqueidentifier también debe incluirse en la lista SELECT.

- Las acciones INSERT que se ejecutan en la vista con particiones deben proporcionar un valor NEWID() para la columna uniqueidentifier. Las acciones UPDATE que se ejecutan en la columna uniqueidentifier deben proporcionar el valor NEWID(), debido a que no se puede usar la palabra clave DEFAULT.

- La duplicación de actualizaciones que se realiza utilizando la vista es exacta a la duplicación de tablas en dos bases de datos distintas; es decir, que agentes de duplicación diferentes dan servicio a las tablas y no se garantiza el orden de las actualizaciones.

Ejemplos

A. Utilizar una instrucción CREATE VIEW sencilla

El ejemplo siguiente crea una vista con una instrucción SELECT sencilla. Una vista sencilla resulta útil cuando una combinación de columnas se consulta frecuentemente.

```
USE pubs
IF EXISTS (SELECT TABLE_NAME FROM INFORMATION_SCHEMA.VIEWS
      WHERE TABLE_NAME = 'titles_view')
   DROP VIEW titles_view
GO
CREATE VIEW titles_view
AS
SELECT title, type, price, pubdate
FROM titles
GO
```

B. Utilizar WITH ENCRYPTION

Este ejemplo utiliza la opción WITH ENCRYPTION y muestra columnas calculadas, columnas con el nombre cambiado y varias columnas.

```
USE pubs
IF EXISTS (SELECT TABLE_NAME FROM INFORMATION_SCHEMA.VIEWS
      WHERE TABLE_NAME = 'accounts')
   DROP VIEW accounts
GO
CREATE VIEW accounts (title, advance, amt_due)
WITH ENCRYPTION
AS
SELECT title, advance, price * royalty * ytd_sales
FROM titles
WHERE price > $5
GO
```

Ésta es la consulta para recuperar el número de identificación y el texto del procedimiento almacenado cifrado:

```
USE pubs
GO
SELECT c.id, c.text
FROM syscomments c, sysobjects o
WHERE c.id = o.id and o.name = 'accounts'
GO
```

El siguiente es el conjunto de resultados:

> **Nota** El resultado de la columna **text** se muestra en una línea separada. Cuando se ejecuta el procedimiento, esta información aparece en la misma línea que la información de la columna **id**.

```
id          text
----------- ------------------------------------------------------------
661577395
??????????????????????????????????????????????????????????????????????..
.

(1 row(s) affected)
```

C. Utilizar WITH CHECK OPTION

En este ejemplo se muestra una vista llamada **CAonly** que permite que la modificación de datos se aplique sólo a los autores del estado de California.

```
USE pubs
IF EXISTS (SELECT TABLE_NAME FROM INFORMATION_SCHEMA.VIEWS
      WHERE TABLE_NAME = 'CAonly')
   DROP VIEW CAonly
GO
CREATE VIEW CAonly
AS
SELECT au_lname, au_fname, city, state
FROM authors
WHERE state = 'CA'
WITH CHECK OPTION
GO
```

D. Utilizar funciones integradas en una vista

Este ejemplo muestra una definición de vista que incluye una función integrada. Cuando se utilizan funciones, la columna que se deriva debe incluir un nombre de columna en la instrucción CREATE VIEW.

```
USE pubs
IF EXISTS (SELECT TABLE_NAME FROM INFORMATION_SCHEMA.VIEWS
```

```
      WHERE TABLE_NAME = 'categories')
   DROP VIEW categories
GO
CREATE VIEW categories (category, average_price)
AS
SELECT type, AVG(price)
FROM titles
GROUP BY type
GO
```

E. Utilizar la función @@ROWCOUNT en una vista

El ejemplo siguiente utiliza la función @@ROWCOUNT como parte de la definición de vista.

```
USE pubs
IF EXISTS (SELECT TABLE_NAME FROM INFORMATION_SCHEMA.VIEWS
      WHERE TABLE_NAME = 'myview')
   DROP VIEW myview
GO
CREATE VIEW myview
AS
   SELECT au_lname, au_fname, @@ROWCOUNT AS bar
   FROM authors
   WHERE state = 'UT'
GO
SELECT *
FROM myview
```

F. Utilizar datos separados

En este ejemplo se utilizan las tablas **SUPPLY1**, **SUPPLY2**, **SUPPLY3** y **SUPPLY4**, que corresponden a las tablas de proveedores de cuatro oficinas ubicadas en distintos países.

```
--create the tables and insert the values
CREATE TABLE SUPPLY1 (
   supplyID INT PRIMARY KEY CHECK (supplyID BETWEEN 1 and 150),
   supplier CHAR(50)
   )
CREATE TABLE SUPPLY2 (
   supplyID INT PRIMARY KEY CHECK (supplyID BETWEEN 151 and 300),
   supplier CHAR(50)
   )
CREATE TABLE SUPPLY3 (
   supplyID INT PRIMARY KEY CHECK (supplyID BETWEEN 301 and 450),
   supplier CHAR(50)
   )
CREATE TABLE SUPPLY4 (
   supplyID INT PRIMARY KEY CHECK (supplyID BETWEEN 451 and 600),
   supplier CHAR(50)
   )
INSERT SUPPLY1 VALUES ('1', 'CaliforniaCorp')
INSERT SUPPLY1 VALUES ('5', 'BraziliaLtd')
INSERT SUPPLY2 VALUES ('231', 'FarEast')
```

```
INSERT SUPPLY2 VALUES ('280', 'NZ')
INSERT SUPPLY3 VALUES ('321', 'EuroGroup')
INSERT SUPPLY3 VALUES ('442', 'UKArchip')
INSERT SUPPLY4 VALUES ('475', 'India')
INSERT SUPPLY4 VALUES ('521', 'Afrique')

--create the view that combines all supplier tables
CREATE VIEW all_supplier_view
AS
SELECT *
FROM SUPPLY1
   UNION ALL
SELECT *
FROM SUPPLY2
   UNION ALL
SELECT *
FROM SUPPLY3
   UNION ALL
SELECT *
FROM SUPPLY4
```

ALTER VIEW

Altera una vista creada previamente (creada al ejecutar CREATE VIEW), incluidas las vistas indizadas, sin afectar a los procedimientos almacenados ni a los desencadenadores dependientes, y sin cambiar los permisos.

Sintaxis

ALTER VIEW [< *database_name* > **.**] [< *owner* > **.**] *view_name* [**(** *column* [**,**...*n*] **)**]
[WITH < view_attribute > [**,**...n]]
AS
   select_statement
[WITH CHECK OPTION]

< view_attribute > ::=
   { ENCRYPTION | SCHEMABINDING | VIEW_METADATA }

Argumentos

view_name

Es la vista que se va a cambiar.

column

Es el nombre de una o más columnas, separadas por comas, que van a formar parte de la vista dada.

Importante Los permisos de columna se mantienen sólo cuando las columnas tienen el mismo nombre antes y después de que se ejecute ALTER VIEW.

Nota En las columnas de la vista, los permisos de un nombre de columna se aplican a través de una instrucción CREATE VIEW o ALTER VIEW, independientemente del origen de los datos subyacentes. Por ejemplo, si se conceden permisos a la columna **title_id** en una instrucción CREATE VIEW, una instrucción ALTER VIEW puede cambiar el nombre de la columna **title_id** (por ejemplo, por **qty**) y seguir teniendo los permisos asociados con la vista mediante **title_id**.

n

Es un marcador de posición que indica que *column* se puede repetir *n* veces.

WITH ENCRYPTION

Cifra las entradas **syscomments** que contienen el texto de la instrucción ALTER VIEW. Utilizar WITH ENCRYPTION evita que la vista se publique como parte de la duplicación de SQL Server.

SCHEMABINDING

Enlaza la vista al esquema. Cuando se especifica SCHEMABINDING, *select_statement* debe incluir el nombre con dos partes (propietario.objeto) de las tablas, vistas o funciones definidas por el usuario a las que se ha hecho referencia.

Las vistas o las tablas que participan en una vista creada con la cláusula de enlace de esquema no se pueden quitar a menos que se quite o cambie esa vista de forma que deje de tener un enlace de esquema. De lo contrario, SQL Server genera un error. Además, las instrucciones ALTER TABLE sobre tablas que participan en vistas que tienen enlaces de esquemas provocarán un error si estas instrucciones afectan a la definición de la vista.

VIEW_METADATA

Especifica que SQL Server devolverá a las API de DBLIB, ODBC y OLE DB la información de metadatos sobre la vista, en vez de las tablas o tabla base, cuando se soliciten los metadatos del modo de exploración para una consulta que hace referencia a la vista. Los metadatos del modo de exploración son metadatos adicionales devueltos por SQL Server a las API DB-LIB, ODBC y OLE DB del cliente, que permiten a las API del cliente implementar cursores actualizables en el cliente. Los metadatos del modo de exploración incluyen información sobre la tabla base a la que pertenecen las columnas del conjunto de resultados.

Para las vistas creadas con la opción VIEW_METADATA, los metadatos del modo de exploración devuelven el nombre de vista en vez de los nombres de la tabla base cuando se describen las columnas de la vista en el conjunto de resultados.

Cuando se crea una vista WITH VIEW_METADATA, todas sus columnas (excepto **timestamp**) son actualizables si la vista tiene los desencadenadores INSERT o UPDATE INSTEAD OF. Consulte las Vistas actualizables en CREATE VIEW.

AS

Son las acciones que va a llevar a cabo la vista.

select_statement

Es la instrucción SELECT que define la vista.

WITH CHECK OPTION

Impone que todas las instrucciones de modificación de datos ejecutadas contra la vista se adhieran a los criterios establecidos en la instrucción especificada en *select_statement* que define la vista.

Si una vista que está actualmente en uso se modifica mediante ALTER VIEW, Microsoft® SQL Server™ impone un bloqueo exclusivo de esquema sobre la vista. Cuando se concede el bloqueo, y no hay usuarios activos de la vista, SQL Server elimina todas las copias de la vista de la memoria caché de procedimientos. Los planes existentes que hacen referencia a la vista permanecen en la memoria caché, pero se vuelven a compilar cuando se llaman.

ALTER VIEW se puede aplicar a vistas indizadas. No obstante, ALTER VIEW quita incondicionalmente todos los índices de la vista.

Ejemplos

A. Alterar una vista

El ejemplo siguiente crea una vista que contiene todos los autores, denominada **All_authors**. Se conceden permisos sobre la vista, pero los requisitos se han cambiado para seleccionar los autores de Utah. A continuación, se utiliza ALTER VIEW para reemplazar la vista.

```
-- Create a view from the authors table that contains all authors.
CREATE VIEW All_authors (au_fname, au_lname, address, city, zip)
AS
SELECT au_fname, au_lname, address, city, zip
FROM pubs..authors
GO
-- Grant SELECT permissions on the view to public.
GRANT SELECT ON All_authors TO public
```

```
GO
-- The view needs to be changed to include all authors
-- from Utah.
-- If ALTER VIEW is not used but instead the view is dropped and
-- re-created, the above GRANT statement and any other statements
-- dealing with permissions that pertain to this view
-- must be re-entered.
ALTER VIEW All_authors (au_fname, au_lname, address, city, zip)
AS
SELECT au_fname, au_lname, address, city, zip
FROM pubs..authors
WHERE state = 'UT'
GO
```

B. Utilizar la función @@ROWCOUNT en una vista

El ejemplo siguiente utiliza la función @@ROWCOUNT como parte de la definición de la vista.

```
USE pubs
GO
CREATE VIEW yourview
AS
    SELECT title_id, title, mycount = @@ROWCOUNT, ytd_sales
    FROM titles
GO
SELECT *
FROM yourview
GO
-- Here, the view is altered.
USE pubs
GO
ALTER VIEW yourview
AS
    SELECT title, mycount = @@ ROWCOUNT, ytd_sales
    FROM titles
    WHERE type = 'mod_cook'
GO
SELECT *
FROM yourview
GO
```

DROP VIEW

Quita una o más vistas de la base de datos actual. DROP VIEW se puede ejecutar en vistas indizadas.

Sintaxis

DROP VIEW { *view* } [**,***...n*]

Argumentos

view

Es el nombre de la vista que se va a quitar. Los nombres de vistas deben seguir las reglas de los identificadores.

n

Es un marcador de posición que indica que se pueden especificar varias vistas.

Observaciones

Cuando quita una vista, la definición y otra información acerca de la vista se eliminan de las tablas del sistema **sysobjects**, **syscolumns**, **syscomments**, **sysdepends** y **sysprotects**. También se eliminan todos los permisos de la vista.

Las vistas de una tabla quitada (mediante la instrucción DROP TABLE) se deben quitar explícitamente con DROP VIEW.

Cuando se ejecuta en una vista indizada, DROP VIEW quita automáticamente todos los índices de una vista. Utilice **sp_helpindex** para mostrar todos los índices de una vista.

Cuando se realiza una consulta a través de una vista, Microsoft® SQL Server™ comprueba que todos los objetos de base de datos a los que se hace referencia en la instrucción existen, que son válidos en el contexto de la instrucción y que las instrucciones de modificación de datos no infringen ninguna regla de integridad de los datos. Las comprobaciones que no son correctas devuelven un mensaje de error. Las comprobaciones correctas convierten la acción en una acción contra las tablas subyacentes.

Si las vistas o tablas subyacentes han cambiado desde que se creó la vista, puede ser útil quitar y volver a crear la vista.

Ejemplos

Este ejemplo quita la vista **titles_view**.

```
USE pubs
IF EXISTS (SELECT TABLE_NAME FROM INFORMATION_SCHEMA.VIEWS
          WHERE TABLE_NAME = 'titles_view')
   DROP VIEW titles_view
GO
```