PROGRAMACIÓN EN TRANSACT-SQL

SQL SERVER 2005

Manual de Referencia para usuarios

Salomón Ccance
CCANCE WEBSITE

PROGRAMACIÓN EN TRANSACT-SQL

Hasta ahora hemos estudiado sentencias SQL orientadas a realizar una determinada tarea sobre la base de datos como definir tablas, obtener información de las tablas, actualizarlas; estas sentencias las hemos ejecutado desde el editor de consultas del MSSMS de una en una o a lo sumo una a continuación de la otra dentro de la misma consulta formando un lote de instrucciones.

Las sentencias que ya conocemos son las que en principio forman parte de cualquier lenguaje SQL. Ahora veremos que TRANSACT-SQL va más allá de un lenguaje SQL cualquiera ya que aunque no permita:

- Crear interfaces de usuario.
- Crear aplicaciones ejecutables, sino elementos que en algún momento llegarán al servidor de datos y serán ejecutados.

Incluye características propias de cualquier lenguaje de programación, características que nos permiten definir la lógica necesaria para el tratamiento de la información:

- Tipos de datos.
- Definición de variables.
- Estructuras de control de flujo.
- Gestión de excepciones.
- Funciones predefinidas.
- Elementos para la visualización, que permiten mostrar mensajes definidos por el usuario gracias a la cláusula PRINT.

Estas características nos van a permitir crear bloques de código orientados a realizar operaciones más complejas. Estos bloques no son programas sino procedimientos o funciones que podrán ser llamados en cualquier momento.

En SQL Server 2005 podemos definir tres tipos de bloques de código, los procedimientos almacenados, los desencadenadores (o triggers) y funciones definidas por el usuario.

Nosotros estudiaremos los dos primeros.

Empezaremos por los procedimientos almacenados, veremos primero las sentencias para crear y eliminar procedimientos almacenados, luego estudiaremos las instrucciones Transact-SQL más propias de un lenguaje de programación nombradas en el tema de Introducción al Transact-SQL como son por ejemplo los bucles y estructuras condicionales. Estas instrucciones las utilizaremos dentro de procedimientos almacenados pero veremos que también nos servirán para definir otros bloques de código.

Terminaremos esta unidad de programación estudiando los disparadores o Triggers muy similares a los procedimientos almacenados que difieren básicamente en la forma en que entran en funcionamiento.

PROCEDIMIENTOS ALMACENADOS (STORE PROCEDURE)

Un procedimiento almacenado (STORE PROCEDURE) está formado por un conjunto de instrucciones Transact-SQL que definen un determinado proceso, puede aceptar parámetros de entrada y devolver un valor o conjunto de resultados. Este procedimiento se guarda en el servidor y puede ser ejecutado en cualquier momento.

Los procedimientos almacenados se diferencian de las instrucciones SQL ordinarias y de los lotes de instrucciones SQL en que están precompilados. La primera vez que se ejecuta un procedimiento, el procesador de consultas de SQL Server lo analiza y prepara un plan de ejecución que se almacena en

una tabla del sistema. Posteriormente, el procedimiento se ejecuta según el plan almacenado. Puesto que ya se ha realizado la mayor parte del trabajo de procesamiento de consultas, los procedimientos almacenados se ejecutan casi de forma instantánea por lo que el uso de procedimientos almacenados mejora notablemente la potencia y eficacia del SQL.

SQL Server incorpora procedimientos almacenados del sistema, se encuentran en la base de datos master y se reconocen por su nombre, todos tienen un nombre que empieza por sp_. Permiten recuperar información de las tablas del sistema y pueden ejecutarse en cualquier base de datos del servidor.

También están los procedimientos de usuario, los crea cualquier usuario que tenga los permisos oportunos.

Se pueden crear también procedimiento temporales locales y globales. Un procedimiento temporal local se crea por un usuario en una conexión determinada y sólo se puede utilizar en esa sesión, un procedimiento temporal global lo pueden utilizar todos los usuarios, cualquier conexión puede ejecutar un procedimiento almacenado temporal global. Éste existe hasta que se cierra la conexión que el usuario utilizó para crearlo, y hasta que se completan todas las versiones del procedimiento que se estuvieran ejecutando mediante otras conexiones. Una vez cerrada la conexión que se utilizó para crear el procedimiento, éste ya no se puede volver a ejecutar, sólo podrán finalizar las conexiones que hayan empezado a ejecutar el procedimiento.

Tanto los procedimientos temporales como los no temporales se crean y ejecutan de la misma forma, el nombre que le pongamos indicará de qué tipo es el procedimiento.

Los procedimientos almacenados se crean mediante la sentencia CREATE PROCEDURE y se ejecutan con EXEC (o EXECUTE). Para ejecutarlo también se puede utilizar el nombre del procedimiento almacenado sólo, siempre que sea la primera palabra del lote. Para eliminar un procedimiento almacenado utilizamos la sentencia DROP PROCEDURE.

ELIMINAR PROCEDIMIENTOS ALMACENADOS

Aunque no sabemos todavía crear un procedimiento comentaremos aquí la instrucción para eliminar procedimientos y así podremos utilizarla en los demás ejercicios.

```
    DROP {PROC|PROCEDURE} [nombreEsquema.]nombreProcedimiento [,...n
].
```

Transact-SQL permite abreviar la palabra reservada PROCEDURE por PROC sin que ello afecte a la funcionalidad de la instrucción.
Ejemplos:

```
    DROP PROCEDURE Dice_Hola;
```

Elimina el procedimiento llamado Dice_Hola.

```
    DROP PROC Dice_Hola;
```

Es equivalente, PROC y PROCEDURE indican lo mismo.

Para eliminar varios procedimientos de golpe, indicamos sus nombres separados por comas:

```
    DROP PROCEDURE Dice_Hola,  Ventas_anuales;
```

Elimina los procedimientos Dice_Hola y Ventas_anuales.

CREAR Y EJECUTAR UN PROCEDIMIENTO

Create procedure

Para crear un procedimiento almacenado como hemos dicho se emplea la instrucción CREATE PROCEDURE:

```
    CREATE {PROC|PROCEDURE} [NombreEsquema.]NombreProcedimiento
          [{@parametro tipo} [VARYING] [= valorPredet]  [OUT|OUTPUT] ]
[,...n]
          AS {  <bloque_instrucciones> [ ...n] }[;]
      <bloque_instrucciones> ::=
      {[BEGIN] instrucciones [END] }
```

Las instrucciones CREATE PROCEDURE no se pueden combinar con otras instrucciones SQL en el mismo lote.

Después del verbo CREATE PROCEDURE indicamos el nombre del procedimiento, opcionalmente podemos incluir el nombre del esquema donde queremos que se cree el procedimiento, por defecto se creará en dbo. Ya que Sqlserver utiliza el prefijo sp_ para nombrar los procedimientos del sistema se recomienda no utilizar nombres que empiecen por sp_.

Como se puede deducir de la sintaxis (no podemos indicar un nombre de base de datos asociado al nombre del procedimiento) sólo se puede crear el procedimiento almacenado en la base de datos actual, no se puede crear en otra base de datos.

Si queremos definir un procedimiento temporal local el nombre deberá empezar por una almohadilla (#) y si el procedimiento es temporal global el nombre debe de empezar por ##.

El nombre completo de un procedimiento almacenado o un procedimiento almacenado temporal global, incluidas ##, no puede superar los 128 caracteres. El nombre completo de un procedimiento almacenado temporal local, incluidas #, no puede superar los 116 caracteres.

Transact-SQL permite abreviar la palabra reservada PROCEDURE por PROC sin que ello afecte a la funcionalidad de la instrucción.

```
    CREATE PROC Calcula_comision AS…
```

Es equivalente a

```
    CREATE PROCEDURE calcula_comision  AS…
```

@parametro: representa el nombre de un parámetro. Se pueden declarar uno o más parámetros indicando para cada uno su nombre (debe de empezar por arroba) y su tipo de datos, y opcionalmente un valor por defecto (=valorPredet) este valor será el asumido si en la llamada el usuario no pasa ningún valor para el parámetro. Un procedimiento almacenado puede tener un máximo de 2.100 parámetros.
Los parámetros son locales para el procedimiento; los mismos nombres de parámetro se pueden utilizar en otros procedimientos. De manera predeterminada, los parámetros sólo pueden ocupar el lugar de expresiones constantes; no se pueden utilizar en lugar de nombres de tabla, nombres de columna o nombres de otros objetos de base de datos.

VARYING Sólo se aplica a los parámetros de tipo cursor por lo que se explicará cuando se expliquen los cursores.

OUTPUT | OUT (son equivalentes)
Indica que se trata de un parámetro de salida. El valor de esta opción puede devolverse a la instrucción

EXECUTE que realiza la llamada.

El parámetro variable OUTPUT debe definirse al crear el procedimiento y también se indicará en la llamada junto a la variable que recogerá el valor devuelto del parámetro. 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 indique el nombre del parámetro en la llamada de la forma @parametro=valor.

Procedimiento básico

```
CREATE PROCEDURE  Dice_Hola
  AS
  PRINT 'Hola';
  GO              --  Indicamos GO para cerrar el lote que crea el procedimiento y empezar  otro lote.
  EXEC Dice_Hola;  -- De esta forma llamamos al procedimiento  (se ejecuta).
```

En este caso, como la llamada es la primera del lote (va detrás del GO) podíamos haber obviado la palabra EXEC y haber escrito directamente:

```
Dice_Hola
```

Con un parámetro de entrada (la palabra que queremos que escriba)

```
CREATE PROCEDURE Dice_Palabra  @palabra CHAR(30)
  AS
  PRINT @palabra;
  GO
  EXEC Dice_Palabra 'Lo que quiera';
  EXEC Dice_Palabra 'Otra cosa';
```

Aquí hemos hecho dos llamadas, una con el valor 'Lo que quiera' y otra con el valor 'Otra cosa'.

Con varios parámetros

Si queremos indicar varios parámetros los separamos por comas.

```
USE Biblio;
  --DROP  PROC  VerUsuariosPoblacion;   --La  comentamos  la  primera vez
  GO
  CREATE   PROCEDURE     VerUsuariosPoblacion   @pob   CHAR(30),@pro CHAR(30)
  AS
  SELECT  *  FROM  usuarios  WHERE   poblacion=@pob  AND  provincia  = @pro;
  GO
  EXEC VerUsuariosPoblacion Madrid,  Valencia
```

En la llamada podemos indicar los valores de los parámetros de varias formas, indicando sólo el valor de los parámetros (en este caso los tenemos que indicar en el mismo orden en que están definidos) como en el ejemplo anterior, o bien indicando el nombre del parámetro:

```
EXEC VerUsuariosPoblacion  @pob=Madrid, @pro=Valencia
```


Indicar el nombre del parámetro en la llamada también nos permite indicar los valores en cualquier orden, la siguiente llamada es equivalente a la anterior, hemos invertido el orden y se ejecuta igual:

```
EXEC VerUsuariosPoblacion  @pro=Valencia, @pob=Madrid
```

En este procedimiento todos los parámetros son obligatorios, no deja llamar con un solo parámetro.

```
EXEC VerUsuariosPoblacion Madrid
```

Da error.

Con parámetros opcionales

Para definir un parámetro opcional tenemos que asignarle un valor por defecto en la definición del procedimiento.

```
DROP PROC VerUsuariosPoblacion2;
  GO
  CREATE PROCEDURE VerUsuariosPoblacion2 @pob CHAR(30),@pro
CHAR(30)='Madrid'
  AS
  SELECT * FROM usuarios WHERE poblacion=@pob AND provincia =
@pro;
  GO
  EXEC VerUsuariosPoblacion2 Madrid
```

En este caso, en la llamada sólo hemos indicado un valor, para el primer parámetro, el paramétro opcional no lo hemos indicado y se rellenará con el valor por defecto que indicamos en la definición.

Lo podemos hacer así porque el parámetro opcional es el último de la lista de parámetros. Cuando el parámetro opcional no es el último la llamada tiene que ser diferente, tenemos que nombrar los parámetros en la llamada, al menos a partir del parámetro opcional.

```
DROP PROC VerUsuariosPoblacion3;
  GO
  CREATE PROCEDURE VerUsuariosPoblacion3 @a CHAR(2),@pob
CHAR(30)='Madrid',@pro CHAR(30)
  AS
  SELECT * FROM usuarios WHERE poblacion=@pob AND provincia =
@pro;
  GO
  EXEC VerUsuariosPoblacion3  a,@pro='Madrid'
```

En este caso el parámetro opcional es el segundo de la lista de parámetros, cuando hacemos la llamada no podemos hacer como en otros lenguajes de dejar un hueco:

```
EXEC VerUsuariosPoblacion3  a, ,'Madrid'
```

Da error.
Esta forma da error, debemos utilizar la palabra DEFAULT o indicar el nombre de todos los parámetros que van detrás del opcional.

```
EXEC VerUsuariosPoblacion3  a,DEFAULT,'Madrid'
```

Con parámetros de salida

Un procedimiento almacenado puede también devolver resultados, definiendo el paramétro como OUTPUT o bien utilizando la instrucción RETURN que veremos en el siguiente apartado.
Para poder recoger el valor devuelto por el procedimiento, en la llamada se tiene que indicar una variable donde guardar ese valor.

Ejemplo:

Definimos el procedimiento ultimo_contrato que nos devuelte la fecha en que se firmó el último contrato en una determinada oficina. El procedimiento tendrá pues dos parámetros, uno de entrada para indicar el número de la oficina a considerar y uno de salida que devolverá la fecha del contrato más reciente de entre los empleados de esa oficina.

```
USE Gestion
  GO
  CREATE PROC ultimo_contrato @ofi  INT, @fecha DATETIME OUTPUT
  AS
  SELECT  @fecha=(SELECT  MAX(contrato)    FROM  empleados  WHERE
oficina=@ofi)
  GO
```

Con @fecha DATETIME OUTPUT indicamos que el parámetro @fecha es de salida, el proceso que realice la llamada podrá recoger su valor después de ejecutar el procedimiento.

En la llamada, para los parámetros de salida, en vez de indicar un valor de entrada se indica un nombre de variable, variable que recogerá el valor devuelto por el procedimiento sin olvidar la palabra OUTPUT:

```
DECLARE @ultima AS DATETIME;
EXEC ultimo_contrato 12,@ultima  OUTPUT;
PRINT @ultima;
```

RETURN

RETURN ordena salir incondicionalmente de una consulta o procedimiento, se puede utilizar en cualquier punto para salir del procedimiento y las instrucciones que siguen a RETURN no se ejecutan. Además los procedimientos almacenados pueden devolver un valor entero mediante esta orden.

```
RETURN [expresion_entera]
```

Expresion_entera es el valor entero que se devuelve.

A menos que se especifique lo contrario, todos los procedimientos almacenados del sistema devuelven el valor 0. Esto indica que son correctos y un valor distinto de cero indica que se ha producido un error.
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 @var si @var es NULL), se genera un mensaje de advertencia y se devuelve el valor 0.
Si queremos recoger el valor de estado devuelto por el procedimiento la llamada debe ser distinta, y seguir el siguiente modelo:

```
EXECUTE  @variable_donde_recogemos_estado  =  nombre_procedimiento
@par, ...
```

Con el procedimiento del punto anterior no se puede utilizar esta forma de devolver un resultado porque lo que se devuelve es una fecha, no es un valor entero, pero si quisiéramos definir un procedimiento que nos devuelva el número de empleados de una oficina podríamos hacerlo de dos formas:

De la forma normal con un parámetro de salida:

```
USE Gestion
GO
CREATE PROC trabajadores @ofi INT,  @num INT OUTPUT
AS
SELECT @num=(SELECT COUNT(*) FROM  empleados WHERE oficina=@ofi)
GO
```

Para obtener en la variable @var el resultado devuelto por el procedimiento la llamada sería:

```
DECLARE @var INT;
EXEC trabajadores 12, @var OUTPUT
```

Utilizando return en vez de un parámetro de salida:

```
CREATE PROC trabajadores2 @ofi INT
AS
RETURN (SELECT COUNT(*) FROM  empleados WHERE oficina=@ofi)
GO
```

Para obtener en la variable @var el resultado devuelto por el procedimiento la llamada sería:

```
DECLARE @var INT;
EXEC @var= trabajadores2 12
PRINT @var
```

Nos visualiza el número de trabajadores de la oficina número 12.

INSTRUCCIONES DE CONTROL DE FLUJO

Disponemos de diferentes elementos para el control de flujo, como pueden ser RETURN, IF... ELSE, WHILE, BREAK, CONTINUE, GO TO, EXECUTE, etc. En los siguientes apartados aprenderemos cómo utilizarlos.

IF… ELSE

Proporciona una ejecución condicional, permite ejecutar o no ciertas instrucciones dependiendo de si se cumple o no una determinada condición.

```
IF condicion
   {  sentencia_sql | bloque_sql }
[ ELSE
   {  sentencia_sql | bloque_sql } ]
```

Si la condición se cumple (da como resultado TRUE) se ejecuta la instrucción SQL o bloque de instrucciones que aparecen a continuación de la condición, si la condición no se cumple se ejecutan las sentencias que aparecen después de la palabra ELSE. El bloque ELSE es opcional.

Ejemplo: Si nos queremos guardar en una consulta todos los ejemplos para probarlos en cualquier momento, es conveniente antes de los CREATE PROCEDURE colocar un DROP PROCEDURE para que la instrucción CREATE no dé error si el procedimiento ya existe, pero la primera vez la instrucción DROP PROC nos dará error porque el procedimiento todavía no existe, así que lo mejor es ejecutar el DROP sólo si el procedimiento existe, utilizando la función object_id('nombre_de_objeto','tipo de objeto') que nos devuelve el id del objeto y NULL si el objeto no existe.

```
IF object_id('trabajadores', 'P') IS NOT NULL
        DROP  PROCEDURE trabajadores;
  GO
  CREATE PROC trabajadores….
```


Otro ejemplo. Ahora queremos el procedimiento y si no existe se mandará un mensaje "el procedimiento no existe":

```
IF object_id('trabajadores', 'P') IS NOT NULL
  BEGIN
         PRINT 'Borramos el procedimiento'
         DROP  PROC trabajadores
  END
  ELSE PRINT 'El procedimiento ya  existe'
```

Cuando queremos definir un bloque de instrucciones utilizamos los delimitadores BEGIN..END

Se pueden anidar varias sentencias IF hasta el límite que permita la memoria.

WHILE - BREAK – CONTINUE

Esta instrucción permite definir un bucle que repite una sentencia o bloque de sentencias mientras se cumpla una determinada condición.

```
WHILE condicion
       { sentencia_sql | bloque_sql }
       |  BREAK
       |  CONTINUE
```

Podemos anidar bucles, colocar un bucle WHILE dentro de otro.

BREAK Produce la salida del bucle WHILE más interno. La instrucción BREAK interna sale al siguiente bucle más externo. Todas las instrucciones que se encuentren después del final del bucle interno se ejecutan primero y después se reinicia el siguiente bucle más externo.

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

Por ejemplo, tenemos los siguientes bucles anidados:

```
WHILE condicion1                -- Bucle 1
Instrucciones_1_1
WHILE  condicion2              -- Bucle 2
     Intrucciones2_1
     If  condicion3 BREAK
     Instrucciones2_2
     IF  condicion4 CONTINUE
     Instrucciones2_3
Instrucciones1_2
```

Las instrucciones se ejecutarían en este orden:
Preguntamos por condicion1:
Si condicion1 se cumple:
Se ejecuta el bloque Instrucciones1_1
Preguntamos por condicon2:
Si se cumple condicion2:
Se ejecuta el bloque de instrucciones Instrucciones_2_1.
Si condición3 se cumple:
Salimos del bucle 2,
Se ejecuta el bloque Instrucciones1_2
Y se vuelve a preguntar por condicion1.
Si condición3 no se cumple:
Se ejecuta el bloque instrucciones2_2
Si se cumple condicion4:
Saltamos el bloque Instrucciones2_3

Y se vuelve a preguntar por condicion2
Si no se cumple condicion4:
Se ejecuta el bloque Instrucciones2_3
Y se vuelve a preguntar por condicion2
Si condicion2 no se cumple:
Se ejecuta el bloque Instrucciones 1_2
Se vuelve a preguntar por condicion1
Si condicion1 no se cumple:
Hemos terminado

WAITFOR

Bloquea la ejecución de un lote, un procedimiento almacenado o una transacción hasta alcanzar la hora o el intervalo de tiempo especificado, o hasta que una instrucción especificada modifique o devuelva al menos una fila. Nosotros estudiaremos los dos primeros casos.

```
WAITFOR {DELAY  'tiempo_a_transcurrir'
          |TIME 'fechaHora_de_ejecucion'}
```

DELAY permite indicar un período de tiempo especificado (hasta un máximo de 24 horas) que debe transcurrir antes de la ejecución de un lote, un procedimiento almacenado o una transacción.

'tiempo_a_transcurrir' Es el período de tiempo que hay que esperar, 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 lo tanto, no se permite la parte de fecha del valor datetime.

TIME permite indicar la hora especifica a la que se ejecuta el lote, el procedimiento almacenado o la transacción.

'fechaHora_de_ejecucion' Es la hora a la que termina la instrucción WAITFOR por tanto a la que empieza la ejecución de las instrucciones siguientes a WAITFOR. 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 lo tanto, no se permite la parte de fecha del valor datetime.

Cada instrucción WAITFOR tiene un subproceso asociado. Si se especifica un gran número de instrucciones WAITFOR en el mismo servidor, se pueden acumular muchos subprocesos a la espera de que se ejecuten estas instrucciones. SQL Server supervisa el número de subprocesos asociados con las instrucciones WAITFOR y selecciona aleatoriamente algunos de estos subprocesos para salir si el servidor empieza a experimentar la falta de subprocesos.

Ejemplo:

```
PRINT  CONVERT(CHAR(8),Getdate(),108);
WAITFOR DELAY '00:00:03'
PRINT CONVERT(CHAR(8),Getdate(),  108);
WAITFOR DELAY '00:03'
PRINT CONVERT(CHAR(8),Getdate(), 108);
```

Visualiza la hora actual, espera 3 segundos y vuelve a visualizar la hora actual, después espera 3 minutos y vuelve a visualizar la hora actual. Se ha utilizado la función CONVERT con el estilo 108 para que aparezca sólo la hora y con segundos.

GOTO

Altera el flujo de ejecución y lo dirige a una etiqueta.

Las etiquetas se indican mediante un nombre seguido del carácter:

```
GOTO NombreEtiqueta
```

Ejemplo:

```
IF Condicion GOTO etiq
    ----
    Etiq:
    ----
    ----
```

Es una instrucción a evitar porque puede llevar a redactar programas no estructurados.

TRY .. CATCH

La estructura TRY…CATCH implementa un mecanismo de control de errores para Transact-SQL, permite incluir un grupo de instrucciones Transact-SQL en un bloque TRY. Si se produce un error en el bloque TRY, el control se transfiere a otro grupo de instrucciones que está incluido en un bloque CATCH.

```
BEGIN TRY
   {sentencia_sql|bloque_sql}
END TRY
BEGIN CATCH
    [{sentencia_sql|bloque_sql}]
END CATCH [  ; ]
```

Una construcción TRY…CATCH no puede abarcar varios bloques de instrucciones Transact-SQL (varios bloques BEGIN…END de instrucciones Transact-SQL) ni una construcción IF…ELSE.

Si no hay errores en el código incluido en un bloque TRY, cuando la última instrucción de este bloque ha terminado de ejecutarse, el control se transfiere a la instrucción inmediatamente posterior a la instrucción END CATCH asociada. Si hay un error en el código incluido en el bloque TRY, el control se transfiere a la primera instrucción del bloque CATCH asociado. Si la instrucción END CATCH es la última instrucción de un procedimiento almacenado o desencadenador, el control se devuelve a la instrucción que llamó al procedimiento almacenado o activó el desencadenador.

Las construcciones TRY…CATCH se pueden anidar.

Las construcciones TRY…CATCH capturan los errores no controlados de los procedimientos almacenados o desencadenadores ejecutados por el código del bloque TRY. Alternativamente, los procedimientos almacenados o desencadenadores pueden contener sus propias construcciones TRY…CATCH para controlar los errores generados por su código. Por ejemplo, cuando un bloque TRY ejecuta un procedimiento almacenado y se produce un error en éste, el error se puede controlar de las formas siguientes:

- Si el procedimiento almacenado no contiene su propia construcción TRY…CATCH, el error devuelve el control al bloque CATCH asociado al bloque TRY que contiene la instrucción EXECUTE.
- Si el procedimiento almacenado contiene una construcción TRY…CATCH, el error transfiere el control al bloque CATCH del procedimiento almacenado. Cuando finaliza el código del bloque CATCH, el control se devuelve a la instrucción inmediatamente posterior a la instrucción EXECUTE que llamó al procedimiento almacenado.

No se pueden utilizar instrucciones GOTO para entrar en un bloque TRY o CATCH pero se puede utilizar para saltar a una etiqueta dentro del mismo bloque TRY o CATCH, o bien para salir de un bloque TRY o

CATCH.

TRY…CATCH no se puede utilizar en una función definida por el usuario.

Recuperar información sobre errores

En el ámbito de un bloque CATCH, se pueden utilizar las siguientes funciones del sistema para obtener información acerca del error que provocó la ejecución del bloque CATCH:

- ERROR_NUMBER() devuelve el número del error.
- ERROR_SEVERITY() devuelve la gravedad.
- ERROR_STATE() devuelve el número de estado del error.
- ERROR_PROCEDURE() devuelve el nombre del procedimiento almacenado o desencadenador donde se produjo el error.
- ERROR_LINE() devuelve el número de línea de la rutina que provocó el error.
- ERROR_MESSAGE() devuelve el texto completo del mensaje de error. Este texto incluye los valores suministrados para los parámetros reemplazables, como longitudes, nombres de objetos u horas.

Estas funciones devuelven NULL si se las llama desde fuera del ámbito del bloque CATCH. Con ellas se puede recuperar información sobre los errores desde cualquier lugar dentro del ámbito del bloque CATCH. Por ejemplo, en la siguiente secuencia de comandos se muestra un procedimiento almacenado que contiene funciones de control de errores. Se llama al procedimiento almacenado en el bloque CATCH de una construcción TRY…CATCH y se devuelve información sobre el error.

Ejemplo, vamos a definir un procedimiento para calcular el precio unitario a partir de un importe y una cantidad:

```
    -- Verificamos que el procedimiento  que vamos a crear no existe.
      IF OBJECT_ID ('CalculaPrecio', 'P' )  IS NOT NULL
           DROP PROCEDURE CalculaPrecio;
      GO
      -- Creamos el procedimiento
      CREATE PROCEDURE CalculaPrecio @importe  MONEY,@cant INT,@precio
MONEY OUTPUT
      AS
      BEGIN TRY
           SELECT @precio=@importe/@cant;
      END TRY
      BEGIN CATCH
      IF  ERROR_NUMBER() = 8134 SELECT @precio=0
      ELSE   SELECT  ERROR_NUMBER()  as  ErrorNumero,ERROR_MESSAGE()  as
MensajeDeError;
      END CATCH;
      GO
      -- Lo utilizamos
      DECLARE @resultado MONEY
      EXEC CalculaPrecio 1000, 0,  @resultado OUTPUT
      SELECT 'resul', @resultado
```

Si al llamar al procedimiento le pasamos una cantidad igual a cero, la división provocará un error, se pasará el control al bloque CATCH, en este bloque se evalúa si el error corresponde al error de división por cero (8134), si es así el procedimiento devuelve un precio igual a cero, sino se envía un mensaje para avisar del error.

Errores controlados por TRY…CATCH

TRY…CATCH detecta todos los errores de ejecución que tienen una gravedad mayor de 10 y que no

cierran la conexión de la base de datos.

TRY…CATCH no detecta:

- Advertencias o mensajes informativos que tienen gravedad 10 o inferior.
- Errores que tienen gravedad 20 o superior que detienen el procesamiento de las tareas de SQL Server Database Engine en la sesión. Si se produce un error con una gravedad 20 o superior y no se interrumpe la conexión con la base de datos, TRY…CATCH controlará el error.
- Atenciones, como solicitudes de interrupción de clientes o conexiones de cliente interrumpidas.
- Cuando el administrador del sistema finaliza la sesión mediante la instrucción KILL.
- Un bloque CATCH no controla los siguientes tipos de errores cuando se producen en el mismo nivel de ejecución que la construcción TRY…CATCH:
- Errores de compilación, como errores de sintaxis, que impiden la ejecución de un lote.
- Errores que se producen durante la recompilación de instrucciones, como errores de resolución de nombres de objeto que se producen después de la compilación debido a una resolución de nombres diferida.

Estos errores se devuelven al nivel de ejecución del lote, procedimiento almacenado o desencadenador.
En el ejemplo siguiente se muestra cómo la construcción TRY…CATCH no captura un error de resolución de nombre de objeto generado por una instrucción SELECT, sino que es el bloque CATCH el que lo captura cuando la misma instrucción SELECT se ejecuta dentro de un procedimiento almacenado (a un nivel inferior).

```
    USE Gestion;
    GO
    BEGIN TRY
       SELECT * FROM TablaQueNoExiste;
    END TRY
    BEGIN CATCH
       SELECT   ERROR_NUMBER()   as   ErrorNumero,ERROR_MESSAGE()   as
MensajeDeError;
    END CATCH
```

Intentamos ejecutar una SELECT de una tabla que no existe en la base de datos. El error no se captura y el control se transfiere fuera de la construcción TRY…CATCH, al siguiente nivel superior, en este caso salta el mensaje de error del sistema.
Al ejecutar la misma instrucción SELECT dentro de un procedimiento almacenado, el error se producirá en un nivel inferior al bloque TRY y la construcción TRY…CATCH controlará el error.

```
    IF OBJECT_ID  ('TablaInexistente','P') IS NOT NULL
       DROP PROCEDURE TablaInexistente;
    GO
    -- Creamos el procedimiento.
    CREATE PROCEDURE TablaInexistente
    AS
       SELECT * FROM TablaQueNoExiste;
    GO
    -- Utilizamos el procedimiento
    BEGIN TRY
       EXECUTE TablaInexistente
    END TRY
    BEGIN CATCH
       SELECT   ERROR_NUMBER()   as   ErrorNumero,ERROR_MESSAGE()   as
MensajeDeError;
    END CATCH;
```


En este caso como el error se produce dentro del procedimiento, el control del error se devuelve al nivel

superior (el que ha realizado el EXECUTE) por lo que es capturado por el TRY y entra en el bloque CATCH, en vez de que salte el mensaje de error del sistema saldrá el nuestro.

DESENCADENADORES O TRIGGERS

Un desencadenador (o Trigger) es una clase especial de procedimiento almacenado que se ejecuta automáticamente cuando se produce un evento en el servidor de bases de datos.

SQL Server permite crear varios desencadenadores para una instrucción específica.

Según el tipo de evento que los desencadena se clasifican en:

- Desencadenadores DML
- Desencadenadores DDL
- Desencadenadores LOGON

Los desencadenadores DML se ejecutan cuando un usuario intenta modificar datos mediante un evento de lenguaje de manipulación de datos (DML). Los eventos DML son instrucciones INSERT, UPDATE o DELETE de una tabla o vista.

Los desencadenadores DDL se ejecutan en respuesta a una variedad de eventos de lenguaje de definición de datos (DDL). Estos eventos corresponden principalmente a instrucciones CREATE, ALTER y DROP de Transact-SQL, y a determinados procedimientos almacenados del sistema que ejecutan operaciones de tipo DDL.

Los desencadenadores logon se activan en respuesta al evento LOGON que se genera cuando se establece la sesión de un usuario.

Nosotros limitaremos nuestro estudio a los desencadenadores DML.

CREATE TRIGGER

Esta instrucción nos permite definir un trigger:

```
CREATE TRIGGER  [NombreEsquema.]NombreTrigger
  ON {tabla | vista } [,...n] ]
  {FOR|AFTER|INSTEAD OF} {[INSERT][,][UPDATE][,][DELETE]}
  AS sentencia_sql  [;] [,...n ]
```

NombreEsquema es el nombre del esquema al que pertenece el desencadenador DML. Los desencadenadores DML se limitan al esquema de la tabla o vista en la que se crearon.

NombreTrigger es el nombre que le queremos dar al desencadenador. No puede comenzar con los símbolos # o ##.

Tabla | vista es el nombre de la tabla o vista en la que se ejecuta el desencadenador. Sólo se puede hacer referencia a una vista mediante un desencadenador INSTEAD OF. No es posible definir desencadenadores DML en tablas temporales locales o globales.

DELETE INSERT UPDATE Especifica las instrucciones de modificación de datos que activan el desencadenador cuando se intenta ejecutarlas en esta tabla o vista. Se debe especificar al menos una opción. En la definición del desencadenador se permite cualquier combinación de estas opciones, en cualquier orden.

AFTER indica 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 cuando sólo se especifica la palabra clave FOR.
Los desencadenadores AFTER no se pueden definir en las vistas.

INSTEAD OF indica 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 pueden utilizar en vistas actualizables que usan WITH CHECK OPTION.

Los desencadenadores INSTEAD OF DELETE/UPDATE no se permiten en tablas que tengan una clave ajena definida con ON DELETE/UPDATE CASCADE.

CREATE TRIGGER debe ser la primera instrucción en el proceso por lotes.

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

Si se especifica el nombre del esquema del desencadenador hay que calificar también el nombre de la tabla o vista.

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

Un desencadenador está diseñado para comprobar o cambiar los datos en base a una instrucción de modificación o definición de datos; no debe devolver datos al usuario por lo que se aconseja no incluir en un desencadenador instrucciones SELECT que devuelven resultados ni las instrucciones que realizan una asignación variable. Si es preciso que existan asignaciones de variable en un desencadenador, tenemos que utilizar la instrucción SET NOCOUNT al principio del mismo para impedir la devolución de cualquier conjunto de resultados.

Los desencadenadores DML usan las tablas lógicas deleted e inserted. 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. Las tablas deleted e inserted guardan los valores antiguos o nuevos de las filas que la acción del usuario puede cambiar.

Si un desencadenador INSTEAD OF definido en una tabla ejecuta una instrucción en la tabla que normalmente volvería a activarlo, al desencadenador no se lo llama de forma recursiva. En su lugar, la instrucción se procesa como si la tabla no tuviera un desencadenador INSTEAD OF e inicia la cadena de operaciones de restricción y ejecuciones de desencadenadores AFTER. Por ejemplo, si para una tabla se define un desencadenador como INSTEAD OF INSERT, y éste ejecuta una instrucción INSERT en la misma tabla, la instrucción INSERT ejecutada por el desencadenador INSTEAD OF no vuelve a llamar al desencadenador. La instrucción INSERT ejecutada por el desencadenador inicia el proceso que realiza las acciones de restricción y activa cualquier desencadenador AFTER INSERT definido para la tabla.

Si un desencadenador INSTEAD OF definido en una vista ejecuta una instrucción en la vista que normalmente volvería a activarlo, no se llamará el desencadenador de forma recursiva. En su lugar, la instrucción se resuelve a modo de modificaciones en las tablas base subyacentes de la vista. En este caso, la definición de la vista debe cumplir todas las restricciones para una vista actualizable.

Aunque una instrucción TRUNCATE TABLE es en realidad un desencadenador DELETE, no puede activar un desencadenador porque la operación no registra las eliminaciones de fila individuales.

Las siguientes instrucciones Transact-SQL no están permitidas en un desencadenador DML:
ALTER DATABASE CREATE DATABASE DROP DATABASE

Además, las siguientes instrucciones Transact-SQL no se permiten en el cuerpo de un desencadenador DML cuando éste se utiliza en la tabla o vista que es objeto de la acción desencadenadora:
CREATE INDEX ALTER INDEX DROP INDEX DROP TABLE

ALTER TABLE cuando se utiliza para hacer lo siguiente:

- Agregar, modificar o quitar columnas.
- Cambiar particiones.
- Agregar o quitar restricciones de tipo PRIMARY KEY o UNIQUE.

Ejemplo:

```
USE Gestion8
GO
CREATE TRIGGER ActualizaVentasEmpleados
ON pedidos FOR INSERT
AS
UPDATE  empleados SET ventas=ventas+inserted.importe
FROM  empleados, inserted
WHERE  numemp=inserted.rep;
GO
```

Cuando se INSERTe un pedido, entrará en funcionamiento el trigger ActualizaVentasEmpleado y se ejecutarán las instrucciones que aparecen después de AS, en este caso actualizará (UPDATE) la tabla empleados sumará a las ventas del empleado (ventas) el importe del pedido insertado (inserted.importe), y sólo actualizará el empleado cuyo numemp coincida con el campo rep del pedido insertado (WHERE numemp=inserted.rep).

```
-- Ahora comprobamos que funciona
SELECT * FROM empleados WHERE  numemp=108;
INSERT                          INTO                          pedidos
(numpedido,fechapedido,rep,clie,cant,importe,fab,producto)
            VALUES
(123456789,getdate(),108,2103,10,100,'Aci',41001)
SELECT * FROM empleados WHERE  numemp=108;
```

Vemos que al insertar un pedido de 100 € del empleado 108, sus ventas han aumentado en 100€.
ALTER TRIGGER

Permite modificar la definición del desencadenador, no permite cambiar su nombre, para cambiar el nombre de un desencadenador hay que eliminarlo (DROP TRIGGER) y volver a crearlo (CREATE TRIGGER).

```
ALTER TRIGGER [NombreEsquema.]NombreTrigger
  ON {tabla|vista}
  {FOR|AFTER|INSTEAD  OF}  {[INSERT][,][UPDATE][,][DELETE]}  [WITH
APPEND]
  AS sentencia_sql  [;] [,...n ]
```

La sintaxis es similar a la instrucción CREATE TRIGGER.

Ejemplo:

```
ALTER TRIGGER  ActualizaVentasEmpleados
ON pedidos FOR INSERT
AS
UPDATE  empleados SET ventas=ventas+inserted.importe
FROM  empleados, inserted
WHERE numemp=inserted.rep AND  inserted.importe IS NOT NULL;
```

Hemos modificado el desencadenador para que si el importe del pedido es nulo, no haga nada, no actualice con un valor nulo.

DISABLE TRIGGER

En ocasiones puede ser útil inhabilitar temporalmente un desencadenador sin que por ello suponga eliminarlo, para estos casos podemos utilizar la sentencia DISABLE TRIGGER.

```
DISABLE TRIGGER  {[NombreEsquema.]NombreTrigger [,...n] | ALL }
  ON {NombreTablaVista | DATABASE |  ALL SERVER} [;]
```

Ejemplo:

```
DISABLE TRIGGER  ActualizaVentasEmpleado ON pedidos;
```

Deshabilita el desencadenador que hemos creado anteriormente, si después de ejecutar esta sentencia se introduce un nuevo pedido, el empleado correspondiente no se actualizará.
Lo podemos comprobar con:

```
SELECT * FROM empleados WHERE  numemp=108;
  INSERT                          INTO                          pedidos
(numpedido,fechapedido,rep,clie,cant,importe,fab,producto)
                        VALUES
(123456791,getdate(),108,2103,10,300,'Aci',41001)
  SELECT * FROM empleados WHERE  numemp=108;

DISABLE TRIGGER ALL ON pedidos;
```

Deshabilita todos los desencadenadores asociados a la tabla pedidos.

```
DISABLE TRIGGER ALL ON DATABASE;
```

Deshabilita todos los desencadenadores definidos en la base de datos actual.

```
DISABLE TRIGGER ALL ON ALL SERVER;
```

Deshabilita todos los desencadenadores definidos en el servidor.

ENABLE TRIGGER

Con esta instrucción volvemos a habilitar los desencadenadores desactivados por la instrucción anterior.

```
ENABLE TRIGGER {[NombreEsquema.]NombreTrigger  [,...n] | ALL }
  ON {NombreTablaVista | DATABASE |  ALL SERVER} [ ; ]
```

Funciona de la misma manera que DISABLE TRIGGER pero habilita en vez de deshabilitar.

Con este ejemplo lo podemos comprobar:

```
    ENABLE TRIGGER  ActualizaVentasEmpleados ON pedidos;
    SELECT * FROM empleados WHERE  numemp=108;
    INSERT                          INTO                          pedidos
(numpedido,fechapedido,rep,clie,cant,importe,fab,producto)
           VALUES (123456792,getdate(),108,2103,10,400,'Aci',41001)
    SELECT * FROM empleados WHERE  numemp=108;
```

DROP TRIGGER

Para eliminar un desencadenador tenemos la instrucción DROP TRIGGER elimina la definición del desencadenador.

```
DROP TRIGGER  NombreEsquema.NombreTrigger [,...n] [;]
```

Ejemplo:

```
DROP TRIGGER ActualizaVentasEmpleados
```

Elimina el desencadenador ActualizaVentasEmpleados.