

Capitolo 12

Stored procedure

Una volta completato questo capitolo, si sarà in grado di:

- Creare le stored procedure
- Specificare i parametri di input/output
- Utilizzare le variabili
- Creare routine strutturate per la gestione degli errori

In SQL Server, la maggior parte del lavoro ricade sulle solide spalle delle stored procedure. Queste costituiscono il nucleo e l'interfaccia dei database pressoché per ogni applicazione SQL Server usata nel mondo. In questo capitolo è possibile apprendere come creare stored procedure che permettono di gestire un ambiente o fornire l'interfaccia di programmazione necessaria per creare applicazioni di database facilmente gestibili ed efficienti. Tranne ove diversamente indicato, tutti gli esempi di codice di questo capitolo sono disponibili nel file Chapter12\code1.sql.

Creazione delle stored procedure

Ogni istruzione eseguita in SQL Server può essere incapsulata all'interno di una stored procedure. Per semplificare è possibile affermare che una stored procedure non è altro che un programma batch di T-SQL cui è stato assegnato un nome e che è stato memorizzato in un database.

La sintassi generica per creare una stored procedure è la seguente:

```
CREATE { PROC | PROCEDURE } [schema_name.] procedure_name [ ; number ]
    [ { @parameter [ type_schema_name. ] data_type }
      [ VARYING ] [ = default ] [ OUT | OUTPUT ] [READONLY]
    ] [ ,...n ]
    [ WITH <procedure_option> [ ,...n ] ]
    [ FOR REPLICATION ]
AS { <sql_statement> [;][ ...n ] | <method_specifier> } [;]
<procedure_option> ::=
    [ ENCRYPTION ] [ RECOMPILE ] [ EXECUTE AS Clause ]
```

Ciò che distingue una stored procedure da un semplice programma batch di T-SQL sono le dimensioni delle strutture di codice che è possibile impiegare, quali variabili, parametrizzazione, gestione degli errori e costrutti di flusso di controllo.

Inserimento di commenti nel codice

Uno dei segni distintivi del codice ben concepito è l'uso di commenti che permettono di semplificare le future operazioni di manutenzione. T-SQL dispone di due differenti costrutti per il codice di commento:

```
--Questo è una singola riga di commento

/*
Questo è un commento
di più righe
*/
```

Variabili, parametri e codici di ritorno

Variabili

Le variabili sono il mezzo che permette di manipolare, archiviare e passare i dati a una stored procedure e di scambiarli tra stored procedure e funzioni. SQL Server dispone di variabili di tipo locale e globale. Le *variabili locali* sono indicate da un singolo simbolo @, mentre le *variabili globali* sono contrassegnate da @@. Notare che le variabili locali possono essere create, lette e scritte, mentre non è possibile creare le variabili globali o scrivere al loro interno. La tabella 12-1 elenca alcune delle variabili globali più comuni.

TABELLA 12-1 Variabili globali

Variabile globale	Definizione
@@ERROR	Codice di errore dall'ultima istruzione eseguita
@@IDENTITY	Ultimo valore di identità inserito nella connessione
@@ROWCOUNT	Numero di righe interessate dall'ultima istruzione
@@TRANCOUNT	Numero di transazioni aperte all'interno della connessione
@@VERSION	Versione di SQL Server

Le istanze delle variabili vengono create mediante la clausola DECLARE, nella quale vengono specificati nome e tipo di dati della variabile. Una variabile può essere definita utilizzando qualsiasi tipo di dati, a eccezione di *text*, *ntext* e *image*. Ad esempio:

```
DECLARE @intvariable INT,
        @datevariable DATE

DECLARE @tablevar TABLE
(ID INT NOT NULL,
Customer VARCHAR(50) NOT NULL)
```



Nota I tipi di dati *text*, *ntext* e *image* sono obsoleti e non dovrebbero essere utilizzati.

Anche se per creare più istanze di variabili è possibile usare un'unica istruzione *DECLARE*, la creazione dell'istanza di una variabile di tabella deve essere eseguita in un'istruzione *DECLARE* separata.

A una variabile può essere assegnato un valore statico o un singolo valore restituito da un'istruzione *SELECT*. Per assegnare un valore possono essere usate le istruzioni *SET* o *SELECT*; tuttavia, quando si esegue una query per assegnare un valore è necessario utilizzare un'istruzione *SELECT*. *SELECT* è usata anche per restituire il valore di una variabile. Una variabile può essere utilizzata per eseguire calcoli, controllare un'elaborazione o come SARG in una query.

Oltre ad assegnare un valore mediante un'istruzione *SET* o *SELECT*, ciò è possibile anche nel momento in cui viene creata l'istanza della variabile.

```
DECLARE @intvariable    INT = 2,
        @datevariable  DATE = GETDATE(),
        @maxorderdate  DATE = (SELECT MAX(OrderDate) FROM Orders.OrderHeader),
        @counter1      INT,
        @counter2      INT
```

```
SET @counter1 = 1
SELECT @counter2 = -1
```

```
SELECT @intvariable, @datevariable, @maxorderdate, @counter1, @counter2
```

Utilizzando le istruzioni *SET* o *SELECT* è possibile eseguire calcoli con le variabili. SQL Server 2008 introduce un modo più compatto di assegnare i valori alle variabili utilizzando un calcolo.

```
--SQL Server 2005 and below
```

```
DECLARE @var    INT

SET @var = 1
SET @var = @var + 1
SELECT @var
SET @var = @var * 2
SELECT @var
SET @var = @var / 4
SELECT @var
GO
```

```
--SQL Server 2008
```

```
DECLARE @var    INT

SET @var = 1
SET @var += 1
SELECT @var
SET @var *= 2
SELECT @var
SET @var /= 4
SELECT @var
GO
```



Importante Tutte le variabili contengono un singolo valore, a eccezione delle variabili di tabella. Anche se è possibile assegnare a una variabile il risultato di un'istruzione *SELECT*, il sistema non restituisce alcun errore qualora tale istruzione restituisca più valori. In questo caso, la variabile conterrà solo l'ultimo valore del set di risultati, mentre tutti gli altri valori saranno eliminati.

Parametri

I parametri sono variabili locali usate per passare valori a una stored procedure quando questa viene eseguita. Durante l'esecuzione, i parametri vengono usati esattamente come le variabili e possono essere letti e scritti.

```
CREATE PROCEDURE <procedure name> @parm1 INT, @parm2 VARCHAR(20) = 'Default value'
AS
    --Code block
```

È possibile creare due tipi di parametri: di input e di output. I parametri di output sono indicati mediante la parola chiave *OUTPUT*.

```
CREATE PROCEDURE <procedure name> @parm1 INT, @parm2 VARCHAR(20) = 'Default value',
    @orderid INT OUTPUT
AS
    --Code block
```

I parametri di output sono usati quando è necessario restituire un singolo valore a un'applicazione. Per restituire un intero set di risultati, è necessario includere nella stored procedure un'istruzione *SELECT* che generi i risultati e restituisca il set di risultati all'applicazione.

```
CREATE PROCEDURE <procedure name> @parm1 INT, @parm2 VARCHAR(20) = 'Default value'
AS
    --This will return the results of this query to an application
    SELECT OrderID, CustomerID, OrderDate, SubTotal, TaxAmount, ShippingAmount, GrandTotal
    FROM Orders.OrderHeader
```

Codici di ritorno

Il codice di ritorno può essere restituito a un'applicazione al fine di determinare lo stato di esecuzione della procedura. I codici di ritorno non sono progettati per inviare i dati, ma solo per segnalare lo stato dell'esecuzione.

```
CREATE PROCEDURE <procedure name> @parm1 INT, @parm2 VARCHAR(20) = 'Default value'
AS
    --This will return the value 1 back to the caller
    RETURN 1
```

Esecuzione delle stored procedure

Alle stored procedure è possibile accedere mediante un'istruzione *EXEC*. Quando una stored procedure non prevede parametri di input, l'unico codice richiesto è il seguente:

```
EXEC <stored procedure>
```

Se una stored procedure dispone di parametri di input, questi possono essere passati utilizzandone il nome o la posizione.

```
--Execute by name
EXEC <stored procedure> @parm1=<value>, @parm2=<value>, ...
--Execute by position
EXEC <stored procedure> <value>, <value>, ...
```

L'esecuzione di una stored procedure per posizione produce un codice più compatto, ma maggiormente soggetto a errori. Quando viene modificato l'ordine dei parametri di una procedura, ciò non ha alcun effetto sul codice quando questa viene eseguita passando i parametri per nome.

Per utilizzare un parametro di output, è necessario specificare le parole chiave *OUT* o *OUTPUT* dopo ogni parametro.

```
--Using output parameters
DECLARE @variable1    <data type>,
        @variable2    <data type>
        ...
EXEC <stored procedure> @parameter1, @variable1 OUTPUT, @variable2 OUT
```

Per catturare il codice restituito da una stored procedure, è necessario salvarlo in una variabile come mostrato di seguito:

```
--Capturing a return code
DECLARE @variable1    <data type>,
        @variable2    <data type>,
        @returncode   INT
EXEC @returncode = <stored procedure> @parameter1, @variable1 OUTPUT, @variable2 OUT
```

Costrutti di flusso di controllo

Le stored procedure dispongono di numerosi costrutti di flusso di controllo utilizzabili:

- *RETURN*
- *IF...ELSE*
- *BEGIN...END*
- *WHILE*
- *BREAK/CONTINUE*

- *WAITFOR*
- *GOTO*

RETURN permette di terminare l'esecuzione della procedura e restituire il controllo all'applicazione chiamante. Il codice che segue l'istruzione *RETURN* non viene eseguito.

```
CREATE PROCEDURE <procedure name> @parm1 INT, @parm2 VARCHAR(20) = 'Default value'
AS
    --This will return the value 1 back to the caller
    RETURN 1

    --Any code from this point on will not be executed
```

IF...ELSE consente di eseguire il codice in modo condizionale. L'istruzione *IF* verifica la condizione fornita ed esegue il blocco di codice successivo se tale condizione è vera. L'istruzione facoltativa *ELSE* permette di eseguire il codice quando la condizione restituisce un risultato falso.

```
DECLARE @var INT

SET @var = 1

IF @var = 1
    PRINT 'This is the code executed when true.'
ELSE
    PRINT 'This is the code executed when false.'
```

Indipendentemente dal percorso seguito dal codice per il costrutto *IF...ELSE*, viene eseguita in modo condizionato solo l'istruzione successiva.

```
DECLARE @var INT

SET @var = 1

IF @var = 2
    PRINT 'This is the code executed when true.'
    PRINT 'This will always execute.'
```

Dal momento che l'istruzione *IF* esegue in modo condizionato solo la riga successiva di codice, si verifica un problema ogni volta che si desidera eseguire un intero blocco di codice mediante la verifica di una condizione. Il costrutto *BEGIN...END* permette di delimitare i blocchi di codice che dovrebbero essere eseguiti come un'unità.

```
DECLARE @var INT

SET @var = 1

IF @var = 2
BEGIN
    PRINT 'This is the code executed when true.'
    PRINT 'This code will also execute only when the conditional is true.'
END
```



Nota Uno degli errori più gravi in cui è possibile incorrere durante la stesura di blocchi di codice che utilizzano un costrutto IF o WHILE è dimenticare che SQL Server esegue in modo condizionale solo l'istruzione successiva. Per evitare gli errori di codifica più comuni, si raccomanda di utilizzare sempre un costrutto BEGIN...END con IF o WHILE, anche quando si desidera eseguire in modo condizionato una sola riga di codice. Ciò non solo rende il codice più leggibile, ma impedisce anche errori in caso di future modifiche del codice.

WHILE permette di eseguire in modo iterativo un blocco di codice se la condizione specificata è vera.

```
DECLARE @var1 INT,
        @var2 VARCHAR(30)

SET @var1 = 1

WHILE @var1 <= 10
BEGIN
    SET @var2 = 'Iteration #' + CAST(@var1 AS VARCHAR(2))

    PRINT @var2

    SET @var1 += 1
END
```

BREAK è usato con un ciclo WHILE. Per terminare l'esecuzione all'interno di un ciclo WHILE, è possibile utilizzare l'istruzione *BREAK* per terminare l'iterazione del ciclo. Una volta eseguita l'istruzione *BREAK*, l'esecuzione del codice prosegue con la riga di codice successiva al ciclo WHILE. Usato all'interno di un ciclo WHILE, *CONTINUE* fa in modo che l'esecuzione del codice continui all'interno del ciclo stesso.



Nota Le istruzioni *BREAK/CONTINUE* non vengono utilizzate quasi mai. Il ciclo *WHILE* termina non appena la relativa condizione non è più vera. Invece di incorporare un test condizionale con un'istruzione *BREAK*, generalmente i cicli *WHILE* sono controllati mediante l'uso di una condizione appropriata per il costrutto *WHILE*. Se la condizione del costrutto *WHILE* è vera, l'esecuzione del ciclo prosegue. Pertanto, non dovrebbe mai essere necessario utilizzare l'istruzione *CONTINUE*.

WAITFOR permette di interrompere momentaneamente l'esecuzione del codice. WAITFOR ha tre diverse permutazioni: WAITFOR DELAY, WAITFOR TIME e WAITFOR RECEIVE. WAITFOR RECEIVE è usato insieme a Service Broker, come illustrato nel capitolo 16, "Service Broker". WAITFOR TIME interrompe temporaneamente l'esecuzione del codice fino all'ora specificata. WAITFOR DELAY interrompe temporaneamente l'esecuzione del codice per l'intervallo di tempo specificato.

```
DECLARE @var1 INT,
        @var2 VARCHAR(30)

SET @var1 = 1

--Pause for 2 seconds
WAITFOR DELAY '00:00:02'
```

```

WHILE @var1 <= 10
BEGIN
    SET @var2 = 'Iteration #' + CAST(@var1 AS VARCHAR(2))

    PRINT @var2

    SET @var1 += 1
END

```

GOTO permette di passare l'esecuzione a un'etichetta incorporata nella procedura. Notare che l'uso di costrutti di codice come GOTO è sconsigliato in qualsiasi linguaggio di programmazione.

Gestione degli errori

In un mondo ideale, il codice del programma eseguito dovrebbe funzionare sempre senza errori. In realtà, il codice sarà sempre soggetto a malfunzionamenti. Pertanto, nelle stored procedure deve essere inclusa la gestione degli errori.

Prima di SQL Server 2005, l'unico modo per gestire gli errori consisteva nel controllare il valore della variabile globale @@error. Grazie all'uso di un blocco *TRY...CATCH*, ora è disponibile una gestione degli errori strutturata simile a quella presente in altri linguaggi di programmazione.

Il blocco TRY...CATCH è formato da due componenti. Il blocco TRY permette di racchiudere il codice da cui è possibile ricevere un errore che si desidera rilevare e gestire. Il blocco CATCH serve a gestire l'errore.

Il codice che segue crea un errore dovuto alla violazione di un vincolo di chiave primaria. Apparentemente, questo codice lascia dietro di sé una tabella vuota a causa dell'errore nella transazione; è evidente tuttavia che la prima e la terza istruzione INSERT vengono eseguite correttamente e inseriscono due righe nella tabella.

```

--Transaction errors

CREATE TABLE dbo.mytable
(ID          INT          NOT NULL PRIMARY KEY)

BEGIN TRAN
    INSERT INTO dbo.mytable VALUES(1)
    INSERT INTO dbo.mytable VALUES(1)
    INSERT INTO dbo.mytable VALUES(2)
COMMIT TRAN

SELECT * FROM dbo.mytable

TRUNCATE TABLE dbo.mytable

```

Il motivo per cui nella tabella vengono inserite due righe è che per impostazione predefinita SQL Server non annulla la transazione che causa l'errore. Per fare in modo che la transazione sia interamente completata o respinta, è possibile utilizzare il comando *SET* per modificare l'impostazione XACT_ABORT della connessione, come mostrato di seguito:

```

SET XACT_ABORT ON;
BEGIN TRAN
    INSERT INTO dbo.mytable VALUES(1)
    INSERT INTO dbo.mytable VALUES(1)
    INSERT INTO dbo.mytable VALUES(2)
COMMIT TRAN
SET XACT_ABORT OFF;

SELECT * FROM dbo.mytable

```

Anche se l'istruzione *SET* permette di raggiungere lo scopo prefisso, se il codice non reimposta correttamente le opzioni, nell'applicazione possono verificarsi risultati imprevedibili quando si modificano le impostazioni di una connessione. Una soluzione migliore consiste nell'utilizzare un gestore degli errori strutturato per rilevare gli errori e decidere come gestirli.

```

--TRY...CATCH
TRUNCATE TABLE dbo.mytable

BEGIN TRY
    BEGIN TRAN
        INSERT INTO dbo.mytable VALUES(1)
        INSERT INTO dbo.mytable VALUES(1)
        INSERT INTO dbo.mytable VALUES(2)
    COMMIT TRAN
END TRY

BEGIN CATCH
    ROLLBACK TRAN
    PRINT 'Catch'
END CATCH

SELECT * FROM dbo.mytable

```

Oltre a fornire una routine di gestione degli errori strutturata, in questo modo si eliminano anche i codici di errore irrecuperabili che possono causare il blocco imprevisto del codice. Come è possibile notare nel precedente esempio di codice, il blocco *CATCH* rileva l'errore mentre questo è ancora attivo, gestisce il problema e restituisce il controllo senza visualizzare il messaggio visto nei due blocchi di codice precedenti.

Esecuzione dinamica

Nonostante l'esecuzione dinamica dei comandi sia impiegata molto raramente all'interno delle stored procedure usate dalle applicazioni, molte procedure amministrative devono poter creare comandi da eseguire in modo dinamico. T-SQL mette a disposizione due modi per eseguire istruzioni create dinamicamente: `EXEC(<comando>)` e `sp_executesql <comando>`.

```

EXEC('SELECT OrderID, CustomerID FROM Orders.OrderHeader WHERE OrderID = 1')
GO

DECLARE @var    VARCHAR(MAX)
SET @var = 'SELECT OrderID, CustomerID FROM Orders.OrderHeader WHERE OrderID = 1'
EXEC(@var)
GO

```

```
EXEC sp_executesql N'SELECT OrderID, CustomerID FROM Orders.OrderHeader WHERE OrderID = 1'
GO

DECLARE @var NVARCHAR(MAX)
SET @var = 'SELECT OrderID, CustomerID FROM Orders.OrderHeader WHERE OrderID = 1'
EXEC sp_executesql @var
GO
```



Importante Ogni volta che si procede alla creazione di una stringa per l'esecuzione dinamica si verifica la possibilità di un attacco SQL injection. Questo tipo di attacco esula dallo scopo di questo manuale, ma prima di scrivere codice che sfrutta i vantaggi dell'esecuzione dinamica si consiglia di leggere i molti articoli pubblicati in merito e di comprendere i rischi relativi.

Cursori

SQL Server è stato creato per elaborare insiemi di dati. Tuttavia, vi sono casi in cui è necessario elaborare i dati una riga alla volta. I cursori consentono di recuperare un insieme di righe da elaborare una alla volta.



Nota SQL Server è costruito e ottimizzato per eseguire operazioni basate su insiemi. L'uso di un cursore fa in modo che il motore esegua un'elaborazione basata sulle singole righe. Per questo motivo, un cursore non può avere prestazioni simili a quelle di un processo equivalente basato su un insieme.

I cursori sono formati da cinque componenti. *DECLARE* permette di definire l'istruzione *SELECT* che è alla base delle righe presenti nel cursore. *OPEN* fa in modo che l'istruzione *SELECT* sia eseguita e carichi le righe in una struttura di memoria. *FETCH* consente di recuperare una riga alla volta dal cursore. *CLOSE* permette di chiudere l'elaborazione nel cursore. Infine, *DEALLOCATE* consente di rimuovere il cursore e annullare l'allocazione delle strutture di memoria contenenti il set di risultati del cursore.



Nota Dovendo creare un cursore che esegue la stessa operazione su ogni riga recuperata dal cursore stesso, è consigliabile riscrivere il processo in modo da usare una più efficiente operazione basata su un insieme.

La sintassi generica per la dichiarazione di un cursore è la seguente:

```
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 ] ] ]
```

Le seguenti istruzioni mostrano tre modi diversi di dichiarare lo stesso cursore.

```
DECLARE curproducts CURSOR FAST_FORWARD FOR
    SELECT ProductID, ProductName, ListPrice FROM Products.Product
GO
```

```
DECLARE curproducts CURSOR READ_ONLY FOR
    SELECT ProductID, ProductName, ListPrice FROM Products.Product
GO
```

```
DECLARE curproducts CURSOR FOR
    SELECT ProductID, ProductName, ListPrice FROM Products.Product
FOR READ ONLY
GO
```

Una volta dichiarato il cursore, è possibile eseguire un comando OPEN per eseguire l'istruzione *SELECT*.

```
OPEN curproducts
```

Fatto ciò, è necessario recuperare i dati dalla riga del cursore mediante un'istruzione *FETCH*. Quando si esegue l'istruzione *FETCH* per la prima volta, un puntatore viene posizionato in corrispondenza della prima riga del set di risultati del cursore. A ogni esecuzione di un'istruzione *FETCH*, il puntatore del cursore viene avanzato di una riga nel set di risultati fino a raggiungere l'ultima riga presente. Ogni istruzione *FETCH* imposta un valore per la variabile globale @@*FETCH_STATUS*. Generalmente, si utilizza un ciclo *WHILE* per eseguire un'iterazione all'interno del cursore, prelevando una riga a ogni ripetizione del ciclo. L'iterazione del ciclo *WHILE* prosegue fintanto che @@*FETCH_STATUS* è uguale a 0.

```
DECLARE @ProductID      INT,
        @ProductName     VARCHAR(50),
        @ListPrice      MONEY

DECLARE curproducts CURSOR FOR
    SELECT ProductID, ProductName, ListPrice FROM Products.Product
FOR READ ONLY

OPEN curproducts

FETCH curproducts INTO @ProductID, @ProductName, @ListPrice

WHILE @@FETCH_STATUS = 0
BEGIN
    SELECT @ProductID, @ProductName, @ListPrice
    FETCH curproducts INTO @ProductID, @ProductName, @ListPrice
END

CLOSE curproducts
DEALLOCATE curproducts
```



Nota Dovendo scrivere stored procedure che utilizzano cursori, in particolar modo se di più livelli, si consiglia di rivalutare il processo in questione. Probabilmente è possibile sostituire i cursori con un più efficiente processo basato su insiemi.

Procedure CLR

Finora in questo capitolo è stato discusso un solo tipo di stored procedure. Invece di creare una stored procedure utilizzando T-SQL, è possibile scriverne una usando un qualsiasi linguaggio .NET supportato.

La routine è scritta in un linguaggio come C# .NET o Visual Basic .NET e compilata in una libreria DLL che viene quindi caricata in SQL Server. Una volta caricata, è possibile definire una stored procedure per ciascun metodo pubblico presente nella DLL.



Nota Gli oggetti di codice, le funzioni, i trigger e le stored procedure di SQL Server possono essere creati utilizzando una routine CLR.

Creazione di una procedura amministrativa

Il capitolo 6 illustra gli indici e il loro utilizzo. Una delle attività amministrative di routine che devono essere eseguite sui database è la deframmentazione degli indici. Per fare ciò è possibile eseguire una reindicizzazione manuale, ma è molto più semplice incapsulare il codice in una stored procedure da eseguire in modo automatizzato.

La stored procedure che segue riunisce molti degli elementi discussi in questo capitolo, tra cui assegnazione di variabili, parametri, cursori, esecuzione dinamica, cicli WHILE e gestione degli errori strutturata. Il codice che segue è tratto dal file Chapter12\code2.sql.

```
--Create our database for administrative objects
CREATE DATABASE DBAdmin
GO

USE DBAdmin
GO

CREATE PROCEDURE dbo.asp_reindex @database SYSNAME, @fragpercent INT
AS
DECLARE @cmd          NVARCHAR(max),
        @table        SYSNAME,
        @schema       SYSNAME

--Using a cursor for demonstration purposes.
--Could also do this with a table variable and a WHILE loop
DECLARE curtable CURSOR FOR
SELECT DISTINCT OBJECT_SCHEMA_NAME(object_id, database_id) SchemaName,
               OBJECT_NAME(object_id,database_id) TableName
FROM sys.dm_db_index_physical_stats (DB_ID(@database),NULL,NULL,NULL,'SAMPLED')
WHERE avg_fragmentation_in_percent >= @fragpercent
FOR READ ONLY
```

```

OPEN curtable
FETCH curtable INTO @schema, @table

WHILE @@FETCH_STATUS = 0
BEGIN
    SET @cmd = 'ALTER INDEX ALL ON ' + @database + '.' + @schema + '.' + @table +
        ' REBUILD WITH (ONLINE = ON)'
    --Try ONLINE build first, if failure, change to OFFLINE build.
    --Offline rebuild using the ALL keyword is required if the table has XML or
    --    SPATIAL indexes
    --Offline rebuild is also required for tables with indexes on image, text, ntext,
    --    varchar(max), nvarchar(max), varbinary(max), and xml data types.
    --We are using the ALL keyword so that we do not have to change database
    --    context in order to retrieve the index name, since a function does not exist
    --    to get the name outside of the database context for an index. If you need
    --    to maximize the online build operations, you will need to modify this proc
    --    to change context to the database to pick up the index name, check the
    --    index column data types and then substitute the index name for the ALL keyword.
    BEGIN TRY
        EXEC sp_executesql @cmd
    END TRY

    BEGIN CATCH
        BEGIN
            SET @cmd = 'ALTER INDEX ALL ON ' + @database + '.' + @schema + '.' + @table
                + ' REBUILD WITH (ONLINE = OFF)'

            EXEC sp_executesql @cmd
        END
    END CATCH

    FETCH curtable INTO @schema, @table
END

CLOSE curtable
DEALLOCATE curtable
GO

--Test
--Fragmentation percent of 0 will rebuild every index
exec dbo.asp_reindex 'SQL2008SBS',0
exec dbo.asp_reindex 'SQL2008SBSFS',0

```

Guida rapida del capitolo 12

Per	Eeguire le seguenti operazioni
Passare un valore a una stored procedure	Utilizzare il parametro <i>INPUT</i> . I parametri possono essere passati per posizione o per nome. Se per il parametro <i>INPUT</i> è specificato un valore predefinito, il parametro è facoltativo e in sua assenza il programma utilizza il valore predefinito specificato.
Restituire un singolo valore da una stored procedure	Utilizzare un parametro <i>OUTPUT</i> . Ogni procedura memorizzata può avere più parametri <i>OUTPUT</i> . Per restituire un set di risultati è necessario specificare un'istruzione <i>SELECT</i> all'interno della procedura.
Eeguire una stored procedure	Se la procedura è il primo comando di un batch, è sufficiente specificarne il nome. In altro caso è necessario utilizzare l'istruzione <i>EXEC</i> .
Eeguire codice in modo condizionato	Il costrutto <i>IF...ELSE</i> è usato per eseguire l'istruzione o il blocco di istruzioni successivo della procedura. <i>WHILE</i> permette di eseguire più volte un blocco di codice.
Eeguire un comando in modo dinamico	Utilizzare <i>EXEC(<comando>)</i> o <i>sp_executesql <comando></i>
Rilevare e gestire gli errori	Racchiudere il blocco di codice in un blocco <i>TRY...CATCH</i>
Elaborare un set di risultati una riga alla volta	Utilizzare un <i>CURSORE</i>