第6章 使用游标

游标(cursor)是一种数据访问机制，它允许用户访问单独的数据行，而非对整个行集进行操作(通过使用SELECT、UPDATE或者DELETE语句进行)。用户可以通过单独处理每一行逐条收集信息并对数据逐行进行操作；这样，可以降低系统开销和潜在的阻隔情况。用户也可以使用这些数据生成 T-SQL代码并立即执行或输出。从另一种角度来看，游标是用户使用 T-SQL代码可以获得数据集中最紧密的数据的一种方法。

6.1 游标声明

读者可能对游标的基本使用方法，包括其声明已经很熟悉了，因此本节对此阐述较少。在本章中涉及到游标的定义以及使用游标的所有规则。

定义游标

DECLARE语句不仅用于定义代码中所使用的变量，而且也用来定义游标。在SQL Server 7中有两种声明游标的方法。SQL Server同时支持SQL-92和T-SQL两种语法。下面的代码片段显示了SQL-92游标的语法：

```
DECLARE cursor_name [INSENSITIVE] [SCROLL] CURSOR
FOR select_statement
[FOR {READ ONLY¦ UPDATE [OF column_name [,...n]]}]
```

语法所需的部分包括以下的必需项： DECLARE、游标变量、 CURSOR、FOR以及SELECT语句。Cursor_name变量是用户以后引用该游标时使用的名字，它必须符合 SQL Server的标准命名约定。关键字 CURSOR指出了此变量是一个游标类型。 FOR关键字和SELECT语句定义了游标的内容。

- INSENSITIVE —— 该选项将游标数据的一个拷贝放入 tempdb中；这样游标看不到数据的变化。该选项有时也称为快照 (snapshot)或静态游标(static cursor)。不允许直接更新包含在该游标中的数据。
- SCROLL —— 该选项允许用户向前或向后移动游标。如果没有该选项用户每时刻只能向前移动一行。虽然有些局限性，但用户通常只作遍历并收集或者更新数据。
- FOR READ ONLY —— 该选项指出游标是只读的。在默认情况下，包含在游标中的数据是可更新的，通过本选项可以作出限制。 FOR UPDATE指示游标是可更新的，正如上述默认情况所述的那样 —— 那么为什么还要使用 FOR UPDATE选项呢？指定该选项的列列表允许用户指定单个列是可更新的；如果不指定，所有的列都是可更新的。使用该选项的另一个原因是可滚动游标和那些在 SELECT语句中使用ORDER BY短语的游标是只读的，且默认对它们无效；这时用户就要使用 FOR UPDATE选项来更新指定数据。

游标的T-SQL版本提供了一些更多的关键字，并且也有一些不同的功能。 Microsoft官方表示以后将要脱离 T-SQL标准，但它却在 SQL Server 7中增强了T-SQL游标声明。下面是 T-SQL游标的语法：

```
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]]]
```

语法所需的部分和SQL-92中的相同。实际上，如果用户忽略这里列举出的其他选项，则用户使用的就是SQL-97游标。也正是由于这些不同点，使T-SQL游标具有更强大的功能。下面是这些不同的选项及其使用描述。

- LOCAL 关键字LOCAL用于定义游标的范围。LOCAL表示只有调用批、存储过程或触发器能够使用游标。当批、存储过程或触发器完成处理后，游标隐含地被释放。
- GLOBAL 关键字GLOBAL也用于定义游标的范围。当游标声明为GLOBAL时，游标可以进行连接，当创建游标的用户退出系统时，游标隐含地被释放。
- FORWARD_ONLY 关键字FORWARD_ONLY用来声明只能从第一行向最后一行滚动的游标。如果使用FORWARD_ONLY游标，用户只能使用FETCH NEXT来获取游标的数据。
- SCROLL 对于SQL-92游标，SCROLL关键字表示用户可以向任意方向移动游标中的数据。
- STATIC STATIC关键字用于表示SQL Server应该将数据拷贝一份放入tempdb中。如果基本表中的数据被修改，那么这种变化不会反映在游标中。
- KEYSET KEYSET游标在它第一次被打开时，其中的行顺序是固定的。其标识关键字集被拷贝到tempdb数据库中的keyset表中。对非标识关键字的任何改变当游标滚动时就会显示出来。在表中插入的数据不会显示在游标中。如果游标中的一行数据被删除，这时用户提取该数据时，SQL Server返回值为-2的变量@@FETCH_STATUS。
- DYNAMIC DYNAMIC游标是指在取数据的过程中对该游标的任何改变都是可获得的。在不同的提取数据过程中，数据、行的数目、行的顺序都会改变。
- FAST_FORWARD FAST_FORWARD选项赋予FORWARD_ONLY, READ_ONLY游标特定的性能提高。如果选定 FAST_FORWARD选项，就不能再指定 SCROLL, FOR_UPDATE或者FORWARD_ONLY。
- READ_ONLY READ_ONLY选项用来指定该游标中的数据不能改变。这种类型的游标不能在WHERE CURRENT OF语句中引用。
- SCROLL_LOCKS SCROLL_LOCKS选项使SQL Server在读游标中的数据时锁住每一行。它保证对游标的锁定和更新都能成功。
- OPTIMISTIC OPTIMISTIC关键字指定SQL Server在读每一行数据时，将不锁定每一行。在这种情况下，如果在游标打开期间正在被更新或删除的行发生变化，修改操作就会失败。
- TYPE_WARNING TYPE_WARNING选项用来指示SQL Server当指定的游标类型隐式地发生变化时给客户端返回一个错误信息。

用户将会在本章看到游标声明的几个示例，程序清单 6-1是一个简单的游标声明的示例。该游标允许用户对qty大于10的图书的书名和数量进行操作。

程序清单6-1 游标声明的示例

```
DECLARE crTitleSales CURSOR FOR
        SELECT  title, qty
        FROM    titles t
                JOIN sales s ON t.title_id = s.title_id
        WHERE   qty > 10
```

6.2 游标使用规则

本节讨论游标的以下两个方面：游标的范围和游标声明如何作用于其最终形式。先讨论后一方面，即不同类型的SELECT语句对实际创建的游标有何不同影响；还将讨论不同类型的游标如何影响系统性能和其他一些使用规则。游标的范围是讨论如何维持游标，并且在过程结束并返回客户之后如何维持。

6.2.1 游标规则

以下是关于游标的一些规则：

- 前向(不可滚动)游标(forward-only cursor)如果应用于大型表会有较好的性能。
- 前向游标默认为动态的，这将使游标能够快速打开并允许游标更新数据。
- 用户不能在相同批或存储过程中创建的表上声明一个游标。
- 用户可以在相同批或存储过程中创建的临时表上声明一个游标。
- 如果用户动态地创建游标，那么可以在相同批或存储过程中创建的表上创建一个游标。这种类型的游标在其创建之后反映任何对表所作的变化。
- 在声明游标的SELECT语句中不允许出现关键字COMPUTE，FOR BROWSE和INTO。
- 如果使用UNION、DISTINCT、GROUP BY或HAVING；或者使用外联结；或者在SELECT语句中的SELECT列表里包含一个常量表达式；游标将对它们不敏感。
- 声明不敏感的游标也总是只读游标。

程序清单6-2表示了当用户试图更新一个声明为不敏感的游标时会发生什么情况。

程序清单6-2 试图声明一个不敏感、可更新的游标

```
DECLARE crAuthors INSENSITIVE CURSOR FOR
        SELECT * FROM authors
        FOR UPDATE
```

该语句执行失败，并返回如下的错误信息：

```
Msg 16929, Level 16, State 1
Cursor is read only
```

由于声明冲突，游标不会被创建。如果用户不再浏览原始数据，就不能更新数据。尽管这是一个前向游标，但它默认为只读游标并且不能设置为可更新。

这些规则是很浅显的，并且所有的例外已经列举出来了，相信读者容易掌握。下面讨论游标的另一方面。

6.2.2 游标范围

游标存在于整个连接中。前面所声明的游标在整个连接存在期间都是可用的，直到连接

关闭或者游标被破坏。用户如何破坏一个游标？简单地方法是将其释放即可，如程序清单 6-3 所示。

程序清单6-3 声明和释放一个游标

```
DECLARE crAuthors INSENSITIVE CURSOR FOR
    SELECT * FROM authors

/*Other code that uses the cursor*/

DEALLOCATE crAuthors
```

如果用户不释放一个游标，则它不仅仍可以使用，而且只要连接一直维持，它还可以保持打开状态和从中提取数据。例如，用户可以在一个批中声明一个游标，而后打开游标，并从该游标中提取数据。用户可以继续周期地取数。游标维持打开状态直到用户关闭它，并且维持可用状态直到游标被破坏。

6.3 打开、关闭和移动游标

现在读者已熟悉了游标打开与关闭的概念，我们将考察如何实际地访问游标中的数据。用户需要以下几个步骤来获取和遍历数据。下面的几节中会列出完成这一过程所需的语句。

6.3.1 OPEN和CLOSE语句

OPEN和CLOSE语句分别用来为使用打开游标和用完后关闭游标。虽然用户必须总是显式地打开一个游标，但有几种不需要显式地关闭游标的方法。

第一种方法是使用SET语句来选择，该语句使游标在事务 (显式或隐式)结束时关闭。通过在下面代码片段中使用该语句，用户可以确保在存储过程或批结束时游标被关闭。

```
SET CURSOR_CLOSE_ON_COMMIT ON
```

如果用户希望在每次调用服务器时保持游标是关闭的，则可以放心地将该选项置为开启状态。

注意 注意即使用户使用 CLOSE语句或设置SET CURSOR_CLOSE_ON_COMMIT ON选项显式地关闭一个游标。该游标仍是已分配了的。如果用户不再使用它但仍希望保持连接状态，就一定要释放它。

另一种方法是简单地关闭当前连接。该动作不仅关闭游标，而且也将释放游标。如果用户希望维持当前连接，该方法显然不能胜任。

回收游标

众所周知回收有助于节约资源，是一件很好的事情。只因用户可以在整个连接存在期间保持游标处于打开状态，因而就不能回收或再利用游标，尤其是在下面的情况下：用户使用优化程序提示或者使用可重复读(可串行化)事务隔离等级保持锁。其结果是：用户可以冻结数据页面并阻止其发生修改从而创建更多的并发事件，但也降低了性能。在本质上，这种回收游标操作对SQL Server资源产生潜在的坏影响。程序清单 6-4显示了如何创建一个阻塞修改操作的游标。

程序清单6-4 创建一个阻塞修改操作的游标

```
/*Either use the holdlock optimizer hint...*/
DECLARE crAuthors CURSOR FOR
        SELECT * FROM authors (holdlock)

/*...or set the transaction isolation level*/

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

DECLARE crAuthors CURSOR FOR
        SELECT * FROM authors (holdlock)
```

如果程序清单6-4的任一部分用于声明游标，只打开而不关闭游标会在 authors表上留下一个页面和表意向锁。仅有的真正好处可能是用户自己可以使用 UPDATE语句对游标所在的特定行进行操作，从而改变数据。

警告 这种保持锁的方法在游标所在的数据页面的剩余行上也保持一个锁。如果用户打算保持该锁，要尽量保持较短时间；否则就会增加锁的并发性从而将对系统性能产生负面影响。

通常情况下，游标的声明、打开、关闭以及释放都应在同一过程或批内进行。这样用户可以做任何要做的事情并且不必使游标等待。

警告 如果用户的批或过程中途夭折，或者如果用户在释放或关闭游标之前执行了RETURN语句，那么用户很可能使游标处于打开或者已分配状态。

6.3.2 FETCH语句

用户可以声明、打开、关闭和释放一个游标了，最后一点是提取数据。这才是问题的开始。对于初学者而言，本身就可能闹笑话。

FETCH语句是使用游标的关键。下面先了解一下其标准使用方法而后深入研究一些有用的方法和游标的一般应用。

1. FETCH的标准使用

FETCH语句的语法是所有游标语句中最复杂的一种，如程序清单 6-5所示。

程序清单6-5 FETCH语法

```
FETCH
[[NEXT¦ PRIOR¦ FIRST ¦LAST ¦ABSOLUTE {n ¦@n} ¦RELATIVE {n ¦@n}] FROM]
name_of_cursor
[INTO @variable1, @variable2, ...]
```

必需的部分只有FETCH和从中提取数据的游标的名称。这个简单使用还不能体现使用游标的效能。在深入开发使用效能之前，我们先考察语法的其余部分。首先表 6-1列举了几种选项，用于控制提取数据。

表6-1 FETCH命令

命 令	描 述
NEXT	检索游标中的下一行
PRIOR	检索游标中的上一行
FIRST	检索游标中的第一行
LAST	检索游标中的最后一行
ABSOLUTE	使用int, smallint, tinyint类型的值或这些数据类型的变量，检索游标中的第 n个物理行。如果n或@n值为负，则游标从其最后一行开始反向检索第 n个物理行
RELATIVE	使用int, smallint, tinyint类型的值或这些数据类型的变量，检索从当前游标所在行位置开始的第n个相对行。如果n或@n值为负，则游标从当前位置开始反向检索第 n个相对行

如果用户在使用任何一个 FETCH命令时，就必须使用 FROM短语 —— 没有例外情况。语法的最后一部分用于将检索的行数据存储到变量中。变量的数目必须与列的数目相匹配，每个变量必须在尺寸和类型上与来自被选择的列列表的相应列相匹配。

2. 游标阈值

用户如何知道已经完成了游标的移动？SQL Server提供一个全局变量(@@FETCH_STATUS)帮助用户解决这个问题。如果 @@FETCH_STATUS为0，则一切正常；如果它为其他值，则有几种情况需要处理。下面两部分将更详细地解释这个变量。

(1) 移出游标行集范围

移出游标行集的开始或末尾都将导致 @@FETCH_STATUS值为-1。用户可以使用一个WHERE短语来检查 @@FETCH_STATUS的值，从而当没有可检索的行时提取操作就停止。考察程序清单6-6中的代码片段。

程序清单6-6 使用WHERE语句检查@@FETCH_STATUS

```
FETCH NEXT FROM crColumnTypes INTO @chvName, @chvNameType
WHILE (@@FETCH_STATUS <> -1)
BEGIN
    SELECT @chvPrint = '    @' + @chvName + ' ' + @chvNameType
    PRINT @chvPrint
    FETCH NEXT FROM crColumnTypes INTO @chvName, @chvNameType
END
```

只要游标不移出行集的末尾， @@FETCH_STATUS值不为-1；用户就可以一直提取行数据直到到达末尾。注意在进入 WHILE结构之前FETCH是如何执行的。这种方法避免了用户在游标中根本没有行数据时进入循环体。另一个 FETCH(在程序清单最后)在处理数据之后在循环内部执行。这样，如果用户在游标到达末尾时再次移动游标，循环就不会再执行。

(2) 处理删除的行

在游标打开期间，若某行被删除则情况会如何？游标仍会查找该行并将游标指向它 —— 尽管它已被删除。 @@FETCH_STATUS也提供了小心处理这种情况的方法。其值为 -2表示游标已移到了一个原来曾是一行但现在已不存在的位置上了。

提示 如果游标定义为INSENSITIVE，则没有必需检查游标中的丢失行(@@FETCH_STATUS=-2)。因为游标实际上是基本数据的一个副本，行不会真正被删除出游标行集。

如果用户不使用WHILE结构对丢失行进行检查，就会在遇到被删除行时退出循环。用户最好在循环结构内进行检查，如程序清单6-7所示，它是程序清单6-6的改进。

程序清单6-7 检查行是否存在于游标中

```
FETCH NEXT FROM crColumnTypes INTO @chvName, @chvNameType
WHILE (@@FETCH_STATUS <> -1)
BEGIN
    IF (@@FETCH_STATUS <> -2)
    BEGIN
        SELECT @chvPrint = '  @' + @chvName + ' ' + @chvNameType
        PRINT @chvPrint
    END
    FETCH NEXT FROM crColumnTypes INTO @chvName, @chvNameType
END
```

如清单所示，对于行存在性的检查放在循环内部，如果行存在，相应数据用于输出信息；否则，循环跳过该行继续取下一行数据。如果行已经被删除时会怎样？则所有的变量都为NULL值，所以它们对用户没多大用处。

警告 如果@@FETCH_STATUS返回-2，被检索数据所存入的变量值都是NULL，如果当前行中参与选择的列都支持NULL类型并且实际正好都包含NULL值，则所有的变量值也都包含NULL值。因此，这些变量中NULL值的存在并不能确信行已被删除；用户必须使用@@FETCH_STATUS来验证行是否已被删除。

6.4 高级游标使用

本节进一步研究游标是如何帮助执行许多生产任务的。本节主要讨论一个存储过程，该存储过程用来帮助生成一个更新表的存储过程。该过程包含很长的代码，以及一些相当复杂的游标和SELECT语句。因此在解释这些代码细节时，读者应当要有耐心。

为了便于参考和解释，代码的所有行都已进行了编号。

阅读程序清单6-8中的代码，看读者能否理解该过程的行为。较为困难的部分是一些复杂的CASE表达式，其中包括嵌套在SELECT语句内部的CASE表达式；而这些SELECT语句又是游标声明中的一部分。在这个过程中，用户输入要更新的表名称和该表的键。

程序清单6-8 从一个存储过程生成存储过程的代码

```
0001: CREATE PROC prGenerateUpdateProc
0002: @chvTable varchar(30),
0003: @chvKey varchar(30)
0004: AS
0005: SET NOCOUNT ON
0006: DECLARE
0007:       @chvName varchar(30),
0008:       @intType int,
0009:       @chvNameType varchar(255),
```

```
0010:       @chvPrint varchar(255),
0011:       @chvName2 varchar(30),
0012:       @chvNameType2 varchar(255),
0013:       @chvMessage varchar(255),
0014:       @intReturnVal int,
0015:       @chvDBName varchar(30)
0016:
0017:  IF NOT EXISTS
0018:    (SELECT  *
0019:    FROM   sysobjects so
0020:    JOIN   syscolumns sc
0021:         ON so.id = sc.id
0022:    WHERE   so.name = @chvTable
0023:    AND    sc.name = @chvKey)
0024:  BEGIN
0025:       SELECT @intReturnVal = 1
0026:       SELECT @chvMessage = 'Either table ''%s'' or column ''%s''' +
➥ '   does not exist in the database ''%s''.'
0027:    SELECT @chvDBName = DB_NAME()
0028:    RAISERROR (@chvMessage, 10, -1, @chvTable, @chvKey, @chvDBName)
0029:    RETURN @intReturnVal
0030:  END
0031:
0032:  DECLARE crColumnTypes SCROLL CURSOR FOR
0033:       SELECT sc.name AS name,
0034:              st2.name +
0035:              CASE
0036:              WHEN st2.type IN (37,45,39,47)
0037:              THEN '(' + RTRIM(CONVERT(varchar(10),sc.length)) + ') '
0038:       WHEN st2.type IN (55, 63)
0039:         THEN '(' + RTRIM(CONVERT(varchar(10),sc.prec)) + ', '
0040:         + RTRIM(CONVERT(varchar(10),sc.scale)) + ') '
0041:       ELSE ' '
0042:       END +
0043:       CASE sc.status & 8
0044:       WHEN 0 THEN 'NOT NULL'
0045:       WHEN 8 THEN 'NULL'
0046:       END AS type
0047:    FROM  syscolumns sc
0048:       JOIN systypes st ON sc.usertype = st.usertype
0049:       JOIN systypes st2 ON st.type = st2.type
```

```
0050:   WHERE id = OBJECT_ID(@chvTable)
0051:   AND  st2.usertype < 100
0052:   AND  st2.name NOT IN ('sysname','timestamp')
0053:   ORDER BY sc.colid
0054:
0055: DECLARE crColumns SCROLL CURSOR FOR
0056:     SELECT sc.name AS name,
0057:            CASE
0058:            WHEN st2.type IN (37,45) THEN
0059:                 CASE
0060:                 WHEN sc.status & 8 = 8 THEN 1
0061:                 WHEN sc.status & 8 = 0 THEN 8
0062:                  END
0063:            WHEN st2.type IN (39,47) THEN
0064:                 CASE
0065:                 WHEN sc.status & 8 = 8 THEN 2
0066:                 WHEN sc.status & 8 = 0 THEN 9
0067:                 END
0068:            WHEN st2.type IN (38, 106, 108, 109, 110) THEN 3
0069:            WHEN st2.type = 111 THEN 4
0070:            WHEN st2.type IN (48, 52, 55, 56, 59, 60, 62, 63, 122) THEN 5
0071:            WHEN st2.type = 50 THEN 6
0072:            WHEN st2.type IN (58, 61) THEN 7
0073:            END AS type
0074:   FROM  syscolumns sc
0075:            JOIN systypes st2 ON sc.type=st2.type
0076:   WHERE  id = OBJECT_ID(@chvTable)
0077:   AND   st2.usertype < 100
0078:   AND   st2.name NOT IN ('sysname','timestamp')
0079:   ORDER BY sc.colid
0080:
0081: SELECT @chvPrint='CREATE PROC prUpdate' + @chvTable
0082:
0083: PRINT @chvPrint
0084:
0085: OPEN crColumnTypes
0086:
0087: FETCH NEXT FROM crColumnTypes INTO @chvName, @chvNameType
0088: IF (@@fetch_status <> -1)
0089: BEGIN
0090:     WHILE 1 = 1
```

```
0091:      BEGIN
0092:          FETCH RELATIVE 0 FROM crColumnTypes INTO @chvName, @chvNameType
0093:          FETCH RELATIVE 1 FROM crColumnTypes INTO @chvName2, @chvNameType2
0094:          SELECT  @chvName2 = @chvName,
0095:          @chvNameType2 = @chvNameType
0096:          SELECT @chvPrint = '    @' + @chvName2 +
➥ SPACE(34-DATALENGTH(@chvName2)) + @chvNameType2
0097:          IF (@@fetch_status <> -1)
0098:          BEGIN
0099:              SELECT @chvPrint = @chvPrint + ', '
0100:              PRINT @chvPrint
0101:          END
0102:          ELSE
0103:          BEGIN
0104:              PRINT @chvPrint
0105:              BREAK
0106:     END
0107:   END
0108: END
0109:
0110: CLOSE crColumnTypes
0111:
0112: PRINT 'AS'
0113: PRINT ''
0114: PRINT 'DECLARE '
0115:
0116: OPEN crColumnTypes
0117: FETCH NEXT FROM crColumnTypes INTO @chvName, @chvNameType
0118:
0119: WHILE (@@fetch_status <> -1)
0120: BEGIN
0121:   IF (@@fetch_status <> -2)
0122:   BEGIN
0123:     SELECT @chvPrint = '    @' + @chvName + '2' +
0124:       SPACE(33-DATALENGTH(@chvName)) + @chvNameType + ','
0125:     PRINT @chvPrint
0126:   END
0127:   FETCH NEXT FROM crColumnTypes INTO @chvName, @chvNameType
0128: END
0129:
0130: CLOSE   crColumnTypes
```

```
0131:
0132: PRINT '    @intReturnVal          int,'
0133: PRINT '    @chvMessage            varchar(255),'
0134: PRINT '    @inyCount              tinyint'
0135: PRINT ''
0136: PRINT 'SELECT @intReturnVal = 0'
0137: PRINT 'SELECT '
0138:
0139: OPEN crColumnTypes
0140:
0141: FETCH NEXT FROM crColumnTypes INTO @chvName, @chvNameType
0142: IF (@@fetch_status <> -1)
0143: BEGIN
0144:   WHILE 1 = 1
0145:   BEGIN
0146:     FETCH RELATIVE 0 FROM crColumnTypes INTO @chvName, @chvNameType
0147:     FETCH RELATIVE 1 FROM crColumnTypes INTO @chvName2, ➥@chvNameType2
0148:     SELECT  @chvName2 = @chvName,
0149:         @chvNameType2 = @chvNameType
0150:     SELECT @chvPrint = '    @' + @chvName2 + '2 = ' + @chvName2
0151:     IF (@@fetch_status <> -1)
0152:     BEGIN
0153:       SELECT @chvPrint = @chvPrint + ','
0154:       PRINT @chvPrint
0155:     END
0156:     ELSE
0157:     BEGIN
0158:       PRINT @chvPrint
0159:       BREAK
0160:     END
0161:   END
0162: END
0163:
0164: CLOSE crColumnTypes
0165:
0166: SELECT @chvPrint = 'FROM   ' + @chvTable
0167: PRINT @chvPrint
0168: SELECT @chvPrint = 'WHERE   ' + @chvKey + ' = @' + @chvKey
0169: PRINT @chvPrint
0170: PRINT ''
0171: PRINT 'IF @@ROWCOUNT = 0'
```

```
0172:  PRINT 'BEGIN'
0173:  PRINT '   SELECT @intReturnVal = -1'
0174:  SELECT @chvPrint= '   SELECT @cvrMessage = '' ' + @chvTable + ' with ➥Id of '' + '
0175:  PRINT @chvPrint
0176:  SELECT @chvPrint= '   RTRIM(CONVERT(varchar(10), @' + @chvKey + ')) ➥+ '
0177:  PRINT @chvPrint
0178:  PRINT '   '' was not found.'' '
0179:  PRINT '   PRINT @cvrMessage'
0180:  PRINT '   RETURN @intReturnVal'
0181:  PRINT 'END'
0182:  PRINT ''
0183:  PRINT 'SELECT  @inyCount = 0'
0184:  PRINT ''
0185:  PRINT 'BEGIN TRAN'
0186:
0187:  OPEN crColumns
0188:
0189:  FETCH NEXT FROM crColumns INTO @chvName, @intType
0190:
0191:  WHILE (@@fetch_status <> -1)
0192:  BEGIN
0193:    IF (@@fetch_status <> -2)
0194:    BEGIN
0195:      IF @chvName <> @chvKey
0196:      BEGIN
0197:        IF @intType = 1
0198:          SELECT @chvPrint = 'IF COALESCE(@' + @chvName +
0199:            '2, 0x0) <> COALESCE(@' + @chvName + ', 0x0)'
0200:        ELSE IF @intType = 2
0201:          SELECT @chvPrint = 'IF COALESCE(@' + @chvName +
0202:            '2, '' '') <> COALESCE(@' + @chvName + ', '' '')'
0203:        ELSE IF @intType = 3
0204:          SELECT @chvPrint = 'IF COALESCE(@' + @chvName +
0205:            '2, 0) <> COALESCE(@' + @chvName + ', 0)'
0206:        ELSE IF @intType = 4
0207:          SELECT @chvPrint = 'IF COALESCE(@' + @chvName +
0208:            '2, ''1/1/1900'') <> COALESCE(@' +
0209:            @chvName + ', ''1/1/1900'')'
0210:        ELSE
0211:          SELECT @chvPrint = 'IF @' + @chvName + '2 <> @' + @chvName
0212:          PRINT @chvPrint
```

```
0213:         PRINT 'BEGIN'
0214:         PRINT '    SELECT @inyCount = @inyCount + 1'
0215:         SELECT @chvPrint = '    UPDATE ' + @chvTable +
0216:           ' SET ' + @chvName + ' = @' + @chvName
0217:         PRINT @chvPrint
0218:         SELECT @chvPrint = '    WHERE ' + @chvKey +
0219:           ' = @' + @chvKey
0220:         PRINT @chvPrint
0221:         PRINT 'END'
0222:         PRINT ''
0223:       END
0224:     END
0225:     FETCH NEXT FROM crColumns INTO @chvName, @intType
0226:   END
0227:
0228:   CLOSE crColumns
0229:   DEALLOCATE crColumns
0230:   DEALLOCATE crColumnTypes
0231:
0232:   PRINT 'IF @inyCount = 0'
0233:   PRINT 'BEGIN'
0234:   PRINT '   ROLLBACK TRAN'
0235:   PRINT '   SELECT @intReturnVal = 1'
0236:   PRINT '   SELECT @chvMessage =  ''No changes were detected. '' + '
0237:   PRINT '              ''No changes were made.'' '
0238:   PRINT '   PRINT @chvMessage'
0239:   PRINT '   RETURN @intReturnVal'
0240:   PRINT 'END'
0241:   PRINT 'ELSE'
0242:   PRINT 'BEGIN'
0243:   PRINT '    COMMIT TRAN'
0244:   PRINT '    RETURN @intReturnVal'
0245:   PRINT 'END'
0246:
0247:   SET NOCOUNT OFF
0248:
0250:   GO
0251:
```

1. 声明过程和一些变量

第1行到第16行代码是创建存储过程的简单 CREATE语句和代码中使用的一些变量声明。过程有两个自变量：表名称和键字段。第 17行到第31行代码检查并确认传递给该过程的表和

列的存在，如果不存在，则返回一个错误信息。

注意 prGenerateUpdateProc存储过程只能用于这样的表上：这些表都有一个作为行的标识列的字段。

2. 游标

第32行到第52行代码声明了用于该过程的两个游标中的一个。该游标在过程中使用了三次，它选择两列：来自syscolumns表(该表保存每个表的所有列名称)的name列和另一个值，该值决定实际数据类型(如果需要的话，还有长度或精度和比例)。

第36行和37行代码处理char和binary数据类型，并将列长加到该数据类型名称后的一对括号中。第38行到第40行代码处理小数和数值数据类型，并将精度和比例值加到括号中。用户不是得到varchar或numeric作为数据类型，而是得到varchar (10)或numeric (5, 3)。第41行代码不加任何值，它应用于所有其他数据类型(因为它们不需要任何长度信息)。第43行到第46行代码只在字符串尾加上短语=NULL或者什么都不加，用来反映该列为空值。

第47行到第49行的联结是从syscolumns到systypes再到systypes的联结。它用来获得每一列的基本数据类型。ORDER BY短语提供了原始列顺序。

第53行到第77行代码定义了第二个游标。同样选择两个列：列名称和 int值，该值用来反映数据类型的分类，包括其空值。表6-2表示了来自嵌套CASE表达式的返回值。

表6-2 来自程序清单6-8的数据类型分类

值	描 述
1	可为空值的binary和varbinary
2	可为空值的char和varchar
3	可为空值的数值类型(money, int, real, decimal等)
4	可为空值的datetime和smalldatetime
5	不能为空值的数值类型(money, int, real, decimal等)
6	比特(定义为非空)
7	不能为空值的datetime和smalldatetime
8	不能为空值的binary和varbinary
9	不能为空值的char和varchar

随着游标的实际创建和数据的提取，这些值在后面的代码中会陆续出现。尽管值5到9在后面的代码中各不相同，但该表提供了数据类型的一个基本分类，并且可以在其他过程中使用。

3. 输出信息

该过程将所有的信息显示在SQL Server Tool窗口中的结果显示窗格中。第6行代码中的SET NOCOUNT ON语句限制“Rows Affected”消息出现在输出信息中间。变量@chvPrint用于输出所有相关信息，用户可以看到它在SELECT语句中使用，其后经常紧跟着一条PRINT语句。另外，包含静态信息的许多行(确切地说是32行)。考察输出结果可以了解输出的静态行是如何匹配设计的。

4. 使用游标

代码的核心是两个定义的游标如何使用。第一个游标crColumnTypes用于产生存储过程的参数。一个小问题是用户不希望在列表的最后一项上出现逗号，下面的代码片段会更好地理解这一点(这个示例是从authors表中生成的)。

```
CREATE PROC prUpdateauthors
```

```
    @au_id          id ,
    @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
AS
```

注意最后一项，@contract的末尾没有逗号。这从代码生成的角度又提出了一个问题：如何避免将逗号加到最后一项上。一个小技巧是指出游标的最后一行，这样逗号就不会出现在最后一项上了。

(1) 生成参数

第87行代码用于执行初始行数据 FETCH，第88行代码中的IF语句用于检查游标指针是否越过了游标的末尾，第 97行代码也如此。而后执行一条其中的表达式值总为 true的WHILE语句。通过执行一条BREAK语句跳出循环。

第92和93行的两条FETCH语句主要用于检查游标指针是否在最后一行上。第一条语句使用Relative 0短语从当前行(行不移动)选择游标行数据送入@chvName和@chvNameType中。第二个语句提取下一行数据送入 @chvName2和@chvNameType2中；这些值在第 94和95行代码中立即被那些在@chvName和@chvNameType中的数据所代替。

为什么作这个麻烦的操作？现在当前行数据已存储在 @chvName2和@chvNameType2中，并且也前移了一行。如果游标指针处于游标中的最后一行，第一个 FETCH会得到该最后一行数据，而第二个FETCH则会移出游标的末尾。在第 94行代码中存储初始数据，而后检查是否移出了界限。如果在循环再次开始之前游标指针不处于最后一行，逗号将被加到变量@chvPrint的末尾并输出(第99行到第100行代码)。如果在最后一行，第二个 FETCH将会使游标指针移出下界，从而将执行第 104和第105行代码，只输出最后一项且不带逗号，并从循环中跳出。

(2) 生成变量声明

第一个游标在第110行代码处被关闭，并在第 116行到130行代码中再次被打开，生成绝大多数的声明变量(见程序清单6-9)。为什么第二个循环不涉及最后的逗号？用户正在末尾添加额外的DECLARE语句，因此所有的生成行上必须有逗号，并且 PRINT语句会特别留意不让最后的逗号出现(第132行代码)。

程序清单6-9 DECLARE语句的部分结果

```
DECLARE
    @au_id2          id,
    @au_lname2          varchar(40),
    @au_fname2          varchar(20),
    @phone2          char(12),
    @address2          varchar(40),
```

```
    @city2          varchar(20),
    @state2         char(2),
    @zip2           char(5),
    @contract2      bit,
    @intReturnVal   int,
    @chvMessage     varchar(255),
  @inyCount         tinyint
```

(3) 创建SELECT语句

游标在第139行到第164行代码中第三次被使用。与它的第一个实例有相同的机制，该游标创建SELECT语句的列列表部分，创建的SELECT语句检索@chvKey中键字段的当前值。在输出的最后一项中也不包含逗号。程序清单 6-10显示了第139行到第164行代码的最终结果。第166行到第170行代码用于添加FROM和WHERE短语部分。

程序清单6-10 SELECT语句的部分结果

```
SELECT
    @au_id2 = au_id,
    @au_lname2 = au_lname,
    @au_fname2 = au_fname,
    @phone2 = phone,
    @address2 = address,
    @city2 = city,
    @state2 = state,
    @zip2 = zip,
    @contract2 = contract
FROM  authors
WHERE  au_id = @au_id
```

(4) 第二个游标

第176行和第191行代码在结果中加入一些错误陷入代码，第 192行到第231行代码加入代码主体部分。通过使用来自第二个游标的数据类型分类，代码主体生成合适的判断语句来检查已存在的数据是否已改变。如果某字段的值不能为空值，它输出一个简单的判断 (程序清单6-11的第二部分)；如果某字段的值可以为空值，它使用 COALESCE函数和合适的列数据类型的默认数据来创建一个判断语句。由于 notes列是varchar数据类型，它使用一个空串 (")作为默认值。由于orig_price是money数据类型，它使用0作为默认值。

程序清单6-11 第二个游标的部分结果

```
IF @phone2 <> @phone
BEGIN
    SELECT @inyCount = @inyCount + 1
    UPDATE authors SET phone = @phone
    WHERE au_id = @au_id
END
```

```
IF COALESCE(@address2, '') <> COALESCE(@address, '')
BEGIN
        SELECT @inyCount = @inyCount + 1
        UPDATE authors SET address = @address
        WHERE au_id = @au_id
END

IF COALESCE(@address2, '') <> COALESCE(@address, '')
BEGIN
        SELECT @inyCount = @inyCount + 1
        UPDATE authors SET address = @address
        WHERE au_id = @au_id
END
```

5. 清除

剩余的代码用于清除游标，在结果显示窗格中输出一些最终代码，以及将 NOCOUNT重新设置为其初始状态off。

这样就完成了所有工作。该过程总体上是容易理解的，虽然其中有一些很复杂的结构，但细致考察后也不难分析出来。用户可以将此过程作为一个实例或者也可以作为其他用户创建的过程的一部分。认真察看master数据库中的系统存储过程，用户可以从大量的优秀代码中学到很多东西。

6.5 使用游标修改数据

在看完一大堆代码之后，现在可以稍微轻松一下，让我们看一下最后一节中所涉及的根据当前游标位置修改数据的概念。首先，研究如何利用游标更新和删除行。而后考察随机选择一种图书并降价50%的过程 —— 这在标准的UPDATE语句中是无法实现的。

6.5.1 游标不能自动更新和删除行

游标实际上不能自动地对行更新和删除，而只能使用一条UPDATE和DELETE语句来完成。两个语句中的任何一个语句的 WHERE短语都能包含短语 Where Current Of和游标的名称，见程序清单6-12中的示例。

程序清单6-12 使用游标删除一个作者

```
DECLARE crAuthors SCROLL CURSOR FOR
        SELECT  *
        FROM   authors
        FOR UPDATE

OPEN crAuthors

FETCH NEXT FROM crAuthors
FETCH NEXT FROM crAuthors
```

```
FETCH NEXT FROM crAuthors
FETCH NEXT FROM crAuthors
FETCH NEXT FROM crAuthors
FETCH NEXT FROM crAuthors

BEGIN TRAN —Start transaction

DELETE authors WHERE CURRENT OF crAuthors

ROLLBACK TRAN —Undo the delete so you don't have to re-add the row

CLOSE crAuthors

DEALLOCATE crAuthors
```

在打开游标之后，过程提取六个连续行 (每一行都不能破坏任何引用完整性规则)，而后使用DELETE语句和Where Current Of短语删除行。游标用于提供位置，而 DELETE语句进行删除动作。

6.5.2 使用游标做高级更新操作

最后一个代码例程是改变 titles表并创建一个存储过程用于随机地将一种图书的价格降低50%。additional列用于存储原始价格。该过程在选择一种新书名进行降价之前重新设置所有的数据项。使用该过程似乎使任务变得简单，但使用传统的 T-SQL UPDATE语句却无法完成该任务。下面考察一下程序清单 6-13。

程序清单6-13 随机选择一种图书进行价格更新

```
1: ALTER TABLE titles
2: ADD orig_price money NULL
3:
4: GO
5:
6: CREATE PROC prHalfPriceTitle
7: AS
8:
9: DECLARE  @intCount int,
10:         @intRow int,
11:         @chvDiscard varchar(6),
12:         @chvPrint varchar(255)
13:
14: BEGIN TRAN
15:
16: SELECT @intCount = COUNT(*) FROM titles (holdlock)
```

```
17: WHERE orig_price IS NULL AND price IS NOT NULL
18: SELECT @intRow = CONVERT(int, RAND() * @intCount) + 1
19:
20: DECLARE crTitles SCROLL CURSOR FOR
21: SELECT title_id FROM TITLES
22: WHERE orig_price IS NULL and price IS NOT NULL
23:
24: OPEN crTitles
25:
26: FETCH ABSOLUTE @intRow FROM crTitles INTO @chvDiscard
27:
28: IF @@FETCH_STATUS >=0
29: BEGIN
30:         UPDATE titles SET price = orig_price WHERE orig_price IS NOT NULL
31:         UPDATE titles SET orig_price = NULL WHERE orig_price IS NOT NULL
32:         UPDATE titles SET orig_price = price WHERE CURRENT OF crTitles
33:         UPDATE titles SET price = price * .5 WHERE CURRENT OF crTitles
34: END
35: ELSE
36:     PRINT 'No titles to update.'
37:
38: CLOSE crTitles
39: DEALLOCATE crTitles
40:
41: SELECT  title_id,
42:          price,
43:          orig_price,
44:          title
45: FROM   titles
46: WHERE   orig_price IS NOT NULL
47:
48: COMMIT TRAN
49:
50: GO
```

第1行到第5行代码对titles表进行改变并加上orig_price列，用以保存折价书的原价格。第6行到第13行用于创建过程和声明变量。第14行代码创建一个事务，该事务在第48行结束。为什么要使用事务？当用户打算查找当前图书序号，并利用该值随机地选取一本图书时，如果在得到原价格之后且打折操作之前添加或删除书名，就可能导致错误 (选择一个不存在的图书) 或者就不能对新加入的书名对应的图书进行打折。

在第16行和17行代码中，在用户进行打折操作期间共享表锁一直维持锁定状态，同时代码返回原价格。holdlock优化程序暗示控制SQL Server在事务全过程中持有锁。所有的用户可

以读titles表但只有一个可以对它进行更新。

第18行代码产生一个从1到图书总数之间的随机数。第20行到第22行代码定义了一个可滚动游标，用户可以使用它选择要更新的行。第16、17和22行代码使用同一个条件。该条件判断orig_price为NULL值，而price不为NULL值；它避免了对一个包含NULL价格的图书进行更新——NULL的50%仍是NULL，所以没有必要对NULL价格进行改变。

第24行到38行代码使用游标查找要更新的行。FETCH ABSOLUTE语句使用随机数将游标移到随机选择的行上。只要@@FETCH_STATUS的值不为-1或-2，就执行四个更新操作。

第一个更新操作将所有价格重新设置为它们的原始值。第二个设置每一行的orig_price值为NULL。第三个将当前游标行的price列的列值保存到orig_price列中。第四个设置当前游标行的价格值为它的50%。

第39行代码破坏该游标——用户已不再需要它。第41行到第46行代码将改变的行选出并送到客户端。最后，第48行到第50行代码交付游标和事务并结束过程。

6.6 小结

读者已看到了使用游标的一些简单的以及十分复杂的示例。游标确实能简化任务的执行，而使用T-SQL语句则可能要繁锁得多，有时甚至根本不可能实现。只有亲自使用一下游标，才能真正理解其工作机制。另外，最好熟悉一下system表，该表中有许多有用的信息，程序清单6-8中代码的设计是完全依赖于这些信息进行的。