2010

SQL 函数

Via T-SQL

一旦成功地从表中检索出数据，就需要进一步操纵这些数据，以获得有用或有意义的结果。这些要求包括：执行计算与数学运算、转换数据、解析数值、组合值和聚合一个范围内的值等。

Sunny D.D
http://www.cnblogs.com/sunnycoder
2010-8-15

一旦成功地从表中检索出数据，就需要进一步操纵这些数据，以获得有用或有意义的结果。这些要求包括：执行计算与数学运算、转换数据、解析数值、组合值和聚合一个范围内的值等。

下表给出了 T-SQL 函数的类别和描述。

函数类别	作用
聚合函数	执行的操作是将多个值合并为一个值。例如 COUNT、SUM、MIN 和 MAX。
配置函数	是一种标量函数，可返回有关配置设置的信息。
转换函数	将值从一种数据类型转换为另一种。
加密函数	支持加密、解密、数字签名和数字签名验证。
游标函数	返回有关游标状态的信息。
日期和时间函数	可以更改日期和时间的值。
数学函数	执行三角、几何和其他数字运算。
元数据函数	返回数据库和数据库对象的属性信息。
排名函数	是一种非确定性函数，可以返回分区中每一行的排名值。
行集函数	返回可在 Transact-SQL 语句中表引用所在位置使用的行集。
安全函数	返回有关用户和角色的信息。
字符串函数	可更改 char、varchar、nchar、nvarchar、binary 和 varbinary 的值。
系统函数	对系统级的各种选项和对象进行操作或报告。
系统统计函数	返回有关 SQL Server 性能的信息。
文本和图像函数	可更改 text 和 image 的值。

函数的组成

函数的目标是返回一个值。大多数函数都返回一个标量值(scalar value)，标量值代表一个数据单元或一个简单值。实际上，函数可以返回任何数据类型，包括表、游标等可返回完整的多行结果集的类型。本章不准备讨论到这个深度，第 12 章将讲解如何创建和使用用户自定义函数，以返回更复杂的数据。

函数已经存在很长时间了，它的历史比 SQL 还要长。在几乎所有的编程语言中，函数调用的方式都是相同的：

Result=Function()

在 T-SQL 中，一般用 SELECT 语句来返回值。如果需要从查询中返回一个值，就可以把 SELECT 当成输出运算符，而不用使用等号：

SELECT Function()

一个论点

对于 SQL 函数而言，参数表示输入变量或者值的占位符。函数可以有任意个参数，有些参数是必须的，而有些参数是可选的。可选参数通常被置于以逗号隔开的参数表的末尾，以便于在函数调用中去除不需要的参数。

在 SQL Server 在线图书或者在线帮助系统中，函数的可选参数用方括号表示。在下列的 CONVERT()函数例子中，数据类型的 length 和 style 参数是可选的:

CONVERT (data-type [(length)], expression[,style])

可将它简化为如下形式，因为现在不讨论如何使用数据类型：

CONVERT(date_type, expression[,style])

根据上面的定义，CONVERT()函数可接受 2 个或 3 个参数。因此，下列两个例子都是正确的：

```
SELECT CONVERT(Varchar(20), GETDATE())
SELECT CONVERT(Varchar(20), GETDATE(), 101)
```

这个函数的第一个参数是数据类型 Varchar(20)，第 2 个参数是另一个函数 GETDATE()。GETDATE()函数用 datetime 数据类型将返回当前的系统日期和时间。第 2 条语句中的第 3 个参数决定了日期的样式。这个例子中的 101 指以 mm/dd/yyyy 格式返回日期。本章后面将详细介绍 GETDATE()函数。即使函数不带参数或者不需要参数，调用这个函数时也需要写上一对括号，例如 GETDATE()函数。注意在书中使用函数名引用函数时，一定要包含括号，因为这是一种标准形式。

确定性函数

由于数据库引擎的内部工作机制，SQL Server 必须根据所谓的确定性，将函数分成两个不同的组。这不是一种新时代的信仰，只和能否根据其输入参数或执行对函数输出结果进行预测有关。如果函数的输出只与输入参数的值相关，而与其他外部因素无关，这个函数就是确定性函数。如果函数的输出基于环境条件，或者产生随机或者依赖结果的算法，这个函数就是非确定性的。例如，GETDATE()函数是非确定性函数，因为它不会两次返回相同的值。为什么要把看起来简单的事弄得如此复杂呢？主要原因是非确定性函数与全局变量不能在一些数据库编程对象中使用(如用户自定义函数)。部分原因是 SQL Server 缓存与预编译可执行对象的方式。例如，即席查询可以使用任何函数，不过如果打算构建先进的、可重用的编程对象，理解这种区别很重要。

以下这些函数是确定性的：

- AVG()(所有的聚合函数都是确定性的)
- CAST()
- CONVERT()
- DATEADD()
- DATEDIFF()
- ASCII()
- CHAR()
- SUBSTRING()

以下这些函数与变量是非确定性的：

- GETDATE()
- @@ERROR
- @@SERVICENAME
- CURSORSTATUS()
- RAND()

在函数中使用用户变量

变量既可用于输入，也可用于输出。在 T-SQL 中，用户变量以@符号开头，用于声明为特定的数据类型。可以使用 SET 或者 SELECT 语句给变量赋值。以下的例子用于将一个 int 类型的变量@MyNumber 传递给 SQRT()函数：

```
DECLARE @MyNumber int
SET @MyNumber=144
```

```
SELECT SQRT(@MyNumber)
```

结果是 12，即 144 的平方根。

用 SET 给变量赋值

以下例子使用另一个 int 型的变量@MyResult，来捕获该函数的返回值。这个技术类似于过程式编程语言中的函数调用样式，即把 SET 语句和一个表达式结合起来，给参数赋值：

```
DECLARE @MyNumber int, @MyResult int
SET @MyNumber = 144
-- Assign the function result to the variable:
SET @MyResult = SQRT(@MyNumber)
-- Return the variable value
SELECT @MyResult
```

用 SELECT 给变量赋值

使用 SELECT 的另一种形式也可以获得同样的结果。对变量要在赋值前要先声明。使用 SELECT 语句来替代 SET 命令的主要优点是，可以在一个操作内同时给多个变量赋值。执行下面的 SELECT 语句，通过 SELECT 语句赋值的变量就可以用于任何操作了。

```
DECLARE @MyNumber1 int, @MyNumber2 int,
@MyResult1 int, @MyResult2 int
SELECT @MyNumber1 = 144, @MyNumber2 = 121
-- Assign the function result to the variable:
SELECT @MyResult1 = SQRT(@MyNumber1),
@MyResult2 = SQRT(@MyNumber2)
-- Return the variable value
SELECT @MyResult1, @MyResult2
```

上面的例子首先声明了 4 个变量，然后用两个 SELECT 语句给这些变量赋值，而不是用 4 个 SELECT 语句给变量赋值。虽然这些技术在功能上是相同的，但是在服务器的资源耗费上，用一个 SELECT 语句给多个变量赋值一般比用多个 SET 命令的效率要高。将一个甚至多个值选进参数的限制是，对变量的赋值不能和数据检索操作同时进行。这就是上面的例子使用 SELECT 语句来填充变量，而用另外一个 SELECT 语句来检索变量中数据的原因。例如，下面的脚本就不能工作:

```
DECLARE @RestockName varchar(50)
SELECT ProductId
     ,@RestockName = Name + ':' + ProductNumber
FROM Production.Product
```

这个脚本会产生如下错误:

```
消息 141，级别 15，状态 1，第 2 行
向变量赋值的 SELECT 语句不能与数据检索操作结合使用。
```

在查询中使用函数

函数经常和查询表达式结合使用来修改列值。这只需将列名作为参数传递给函数即可，随后函数将引用插入到 SELECT 查询的列的列表中，如下所示：

```
SELECT Title, NationalIDNumber, YEAR(BirthDate) AS BirthYear
FROM HumanResources.Employee
```

在这个例子中，BirthDate 列的值被作为参数传递给 YEAR()函数。函数的结果是别名为 BirthYear 的列。

嵌套函数

我们需要的功能常常不能仅由一个函数来实现。根据设计，函数应尽量简单，用于提供特定的功能。如果一个函数要执行许多不同的操作，就变得复杂和难以使用。因此，每个函数通常仅执行一个操作，要实现所有的功能，可以将一个函数的返回值传递给另一个函数，这称为嵌套函数调用。

以下是一个简单的例子：GETDATE()函数的作用是返回当前的日期与时间，但不能返回经过格式化的数据，因为这是 CONVERT()函数的功能。要想同时使用这两个函数，可以把 GETDATE()函数的输出作为 CONVERT()函数的输入参数。

```
SELECT CONVERT(Varchar(20), GETDATE(), 101)
```

聚合函数

报表的典型用途是从全部数据中提取出代表一种趋势的值或者汇总值，这就是聚合的意义。聚合函数回答数据使用者的如下问题:

上个月鸡雏的总销售量是多少?

19～24 岁之间的巴西男性在食品调味品上的平均支出是多少?

上季度所有订单中从订购到运输的最长时间是多少?

收发室里仍在工作的最老的员工是谁?

聚合函数应用特定的聚合操作并返回一个标量值(单一值)。返回的数据类型对应于该列或者传递到函数中的值。聚合经常和分组、累积以及透视等表运算一起使用，生成数据分析结果。第 7 章将详细介绍这个主题，这里仅讨论简单 SELECT 查询中的一些常用函数。

聚合函数不仅可用在 SELECT 查询中，还可以和标量输入值一起使用。那么，这样做的意义是什么呢？在下列代码中，将值 15 传递给下列聚合函数，每个函数的返回值都相同：

```
SELECT AVG(15)
SELECT SUM(15)
SELECT MIN(15)
SELECT MAX(15)
```

它们都返回 15。虽然，对同一个值求平均、求和、求最小值、求最大值，所得的结果还是那个值。如果对一个值计数，又会产生什么结果呢？

```
SELECT COUNT(15)
```

得到的值是 1，因为函数只计数了一个值。

现在做一些有意义的事。聚合函数只有在处理结果集合中的一组数据时才有意义。每个函数都处理某列的非空值。除非使用分组操作(详见第 7 章)，否则不能在同一个 SELECT 语句中既返回聚合的值，又返回常规的列值。

AVG()函数

AVG()函数用于返回一组数值中所有非空数值的平均值。例如，表 6-2 包含了体操成绩。

表 6-2

体操运动员	项 目	成 绩
Sara	**跳马**	**9.25**
Cassie	跳马	8.75
Delaney	**跳马**	**9.25**
Sammi	跳马	8.05
Erika	**跳马**	**8.60**

Sara	平衡木	9.70
Cassie	**平衡木**	**9.00**
Delaney	平衡木	9.25
Sammi	**平衡木**	**8.95**
Erika	平衡木	8.85

对这些数据执行以下查询:

SELECT AVG(Score)

结果是 8.965。

如果有三个女孩没有完成一些项目，在表中没有记录成绩，则可用 NULL 来表示(见表 6-3)。

表 6-3

体操运动员	项 目	成 绩
Sara	跳马	9.25
Cassie	跳马	8.75
Delaney	跳马	NULL
Sammi	跳马	8.05
Erika	跳马	8.60
Sara	平衡木	9.70
Cassie	平衡木	NULL
Delaney	平衡木	9.25
Sammi	平衡木	NULL
Erika	平衡木	8.85

脚本：

```
create table #GymEvent(Player varchar(10),[Subject] nvarchar(5),Score
decimal(4,2))
go
insert into #GymEvent values('Sara','跳马',9.25)
insert into #GymEvent values('Cassie','跳马',8.75)
insert into #GymEvent values('Delaney','跳马',NULL)
insert into #GymEvent values('Sammi','跳马',8.05)
insert into #GymEvent values('Erika','跳马',8.60)
insert into #GymEvent values('Sara','平衡木',9.70)
insert into #GymEvent values('Cassie','平衡木',NULL)
insert into #GymEvent values('Delaney','平衡木',9.25)
insert into #GymEvent values('Sammi','平衡木',NULL)
insert into #GymEvent values('Erika','平衡木',8.85)
go
drop table #GymEvent
```

在这种情况下，计算平均值时只考虑实际的数值，NULL 不参与运算，结果是 8.921429。但是，如果把缺少的成绩也算在内，即用数值 0 代替 NULL，则会严重影响最终成绩(6.245)，她们能不能进入国家级的比赛就难说了。

COUNT()函数

COUNT()函数用于返回一个列内所有非空值的个数，这是一个整型值。比如，在上一个例子中，体操数据被保存在#GymEvent 表中，要确定 Sammi 参加的项目数，则可以执行下列查询：

```
SELECT COUNT(Score) FROM #GymEvent WHERE Player='Sammi'
```

结果是 1，因为 Sammi 只参加了跳马比赛，她的平衡木成绩是 NULL。

如果需要确定表中的行数，无论这些行是不是 NULL 值，都可以使用以下语法：

```
SELECT COUNT (*) FROM #GymEvent
```

以 Sammi 为例，COUNT(*)查询如下所示:

```
SELECT COUNT(*) FROM #GymEvent WHERE Player='Sammi'
```

由于 COUNT(*)函数会忽略 NULL 值，所以这个查询的结果是 2。

MIN()与 MAX()函数

MIN()函数用于返回一个列范围内的最小非空值；MAX()函数用于返回最大值。这两个函数可以用于大多数的数据类型，返回的值根据对不同数据类型的排序规则而定。为了说明这两个函数，假设有一个表包含了两列值，一列是整型值，另一列是字符型值，如表 6-4 所示。

表 6-4

IntegerColumn(int 类型)	VarCharColumn(varChar 类型)
2	2
4	4
12	12
19	19

脚本：

```
create table #Temp(IntegerColumn int,VarCharColumn varchar(10))
go
insert into #Temp values(2,'2')
insert into #Temp values(4,'4')
insert into #Temp values(12,'12')
insert into #Temp values(19,'19')
go
drop table #Temp
```

如果分别调用 MIN()与 MAX()函数将会返回什么值呢？

```
select MIN(IntegerColumn),MAX(IntegerColumn) from #Temp
select MIN(VarCharColumn),MAX(VarCharColumn) from #Temp
```

(无列名)	(无列名)
2	19

(无列名)	(无列名)
12	4

因为 VarCharColumn 中值的存储类型为字符类型，而不是数字，所以结果以每个字符的 ASCII 值为顺序从左到右排序。这就是 12 比其他值小、而 4 比其他值大的原因。

SUM()函数

SUM()函数是最常用的聚合函数之一，它的功能很容易理解：和 AVG()函数一样，它用于数值数据类型，返回一个列范围内所有非空值的总和。

配置变量

配置变量不是函数，不过它们的用法和系统函数相同。每个全局变量都能够返回 SQL Server 执行环境的标量信息。以下是一些常见的例子。

@@ERROR 变量

这个变量包含当前连接发生的最后一次错误的代码。在执行的语句没有错误时，@@ERROR 变量的值是 0。出现标准错误时，错误是由数据库引擎引发的。所有的标准错误代码与消息都保存在 sys.messages 系统视图中，可以使用如下脚本查询：

```
SELECT * FROM sys.messages
```

定制错误可以通过调用 RAISERROR 语句来手动引发，并调用 sp_addmessage 系统存储过程将其添加到 sysmessages 表中。

以下是一个@@ERROR 变量的简单例子。先试着将一个数除以 0，数据库引擎会引发标准错误号为 8134 的错误。注意查看 Results 选项卡中的查询结果。在发生错误时，Management Studio 的 Messages 选项卡将默认显示在 Results 选项卡的上面：

```
SELECT 5 / 0
SELECT @@ERROR
```

在成功检索@@ERROR 的值后，@@ERROR 的值将返回 0，因为@@ERROR 只保存了上次执行的语句的错误代码。如果希望检索更多的错误信息，可以使用如下脚本从 sysmessages 视图中得到：

```
SELECT 5 / 0
SELECT * FROM master.dbo.sysmessages WHERE error = @@ERROR
```

本节的后面部分内容将说明如何通过使用错误函数来更高效地返回错误数据。

除了美国英语之外，SQL Server 还默认安装了其他语言。每种语言专用的错误消息都有一个语言标识符(mslangid)，对应于 syslanguages 表中的一种语言，如下图所示。

error	severity	dlevel	description	msglangid
8134	16	0	Divide by zero error encountered.	1033
8134	16	0	Fehler aufgrund einer Division durch Null.	1031
8134	16	0	Division par zéro.	1036
8134	16	0	0 除算エラーが発生しました。	1041
8134	16	0	Error de división entre cero.	3082
8134	16	0	Errore di divisione per zero.	1040
8134	16	0	Обнаружена ошибка: деление на ноль.	1049
8134	16	0	Erro de divisão por zero.	1046
8134	16	0	發現除以零的錯誤。	1028
8134	16	0	0 으로 나누기 오류가 발생했습니다.	1042
8134	16	0	遇到以零作除数错误。	2052

属性名 mslangid 被非正式地定义为 Microsoft Global Language Identifier。微软公司用这

个标识符来标识一种语言或语言和国家的组合，微软公司把语言和国家的组合定义为地区。例如，在随 SQL Server 安装的英语中，美国英语的 mslangid 是 1033，英国英语的 mslangid 是 2057。要检索出所有已安装的、支持的语言，可以执行下面的查询：

```
SELECT alias, name, msglangid
FROM sys.syslanguages
```

@@SERVICENAME 变量

这个变量是用于执行和维护当前 SQL Server 实例的 Windows 服务名。它通常返回 SQL Server 默认实例 MSSQLSERVER，但 SQL Server 的指定实例有唯一的服务名。例如在名为 WoodVista 的计算机上有两个 SQL Server 实例：默认实例和指定实例 AughtEight。如在默认实例上检索@@SERVICENAME 全局变量的内容，将返回 MSSQLSERVER，但在指定实例上检索，会返回 AUGHTEIGHT。

@@TOTAL_ERRORS 变量

这个变量用于记录从打开当前连接开始发生的总错误次数。和@@ERROR 变量一样，它对每个用户会话是唯一的，并将在连接关闭时被重置。

@@TOTAL_READ 变量

这个变量记录从打开当前连接时开始计算的磁盘读取总数。DBA 使用这个变量查看磁盘读取活动的情况。

@@VERSION 变量

这个变量包含当前 SQL Server 实例的完整版本信息。

```
SELECT @@VERSION
```

比如，对于运行在 Windows 7 上的 SQL Server 2008 开发版实例，以上脚本能够返回如下信息:

Microsoft SQL Server 2008 (RTM) - 10.0.1600.22 (Intel X86) Jul 9 2008 14:43:34 Copyright (c) 1988-2008 Microsoft Corporation Enterprise Edition on Windows NT 6.1 <X86> (Build 7600:)

实际的版本号是一个简单的整型值，它在微软公司内部使用。而发行的产品可能有其他的商标名。在本例中，SQL Server 2005 的版本是 9，SQL Server 2008 的版本是 10。Windows XP Professional 显示为 Windows NT 5.l 版，而 Vista 显示为 6.0 版。构建号用于内部控制，反映 beta 版和预览版以及正式发行后的补丁包的变化。

错误函数

前面学习了如何使用@@ERROR 全局变量来检索错误信息。而返回所有错误数据的更好方法是使用错误函数。这些函数返回的信息可以存储在错误跟踪表中，以供错误审核。错误函数嵌套在错误处理例程中。第 11 章将详细讨论错误处理，其实通过使用嵌套在 TRY 和 END TRY 语句中的代码块，后跟一个放在 CATCH 和 END CATCH 语句中的代码块就可以实现错误处理。

```
--Try to do something
BEGIN TRY
   SELECT 5 / 0
END TRY
```

```
--If it causes an error, do this
BEGIN CATCH
   PRINT ERROR_MESSAGE()
END CATCH
```

所谓的错误捕获，其实就是这个意思。如果运行上面的示例，将不会出现可识别的错误，因为错误将被捕获并在 CATCH 语句块中进行处理。在编写错误处理代码时，SQL 程序员必须把这些代码放在会引发系统错误的 catch 代码块中。

下列几个错误函数用于返回错误的特定信息：

函数	说明
ERROR_MESSAGE()	返回错误的描述。
ERROR_NUMBER()	返回错误号。
ERROR_SEVERITY()	返回错误的严重级别。错误的严重级别是一个从 0 到 25 的整数。
ERROR_STATE()	返回错误的状态号。错误状态是一个整数，可以唯一地表示系统错误的原因。
ERROR_LINE()	返回例程中导致出错的行号。
ERROR_PROCEDURE()	返回发生错误的存储过程名或触发器名。

下表简要描述了严重级别。

严 重 级 别	说　明
0～10	信息性消息。不会引发系统错误
11～16	用户可以更正的错误，例如违反了外键或主键规则
17	非致命的、不重要的资源错误
18	非致命的内部错误
19	致命的、不重要的资源错误
20	当前进程中的致命错误
21	所有进程中的致命数据库错误
22	致命的表完整性错误
23	致命的数据库完整性错误
24	致命的硬件错误
25	致命的系统错误

下面脚本使用 T-SQL 的内置错误处理功能，来捕获和输出遇到除 0 错误时返回的错误数据。SELECT 命令的结果将显示在 Management Studio 的消息选项卡中。

```
--Try to do something
BEGIN TRY
    SELECT 5 / 0
END TRY
--If it causes an error, do this
BEGIN CATCH
    SELECT ERROR_MESSAGE(),ERROR_NUMBER(),ERROR_SEVERITY(),
        ERROR_STATE(),ERROR_LINE(),ERROR_PROCEDURE()
END CATCH
```

可以看出，执行这个脚本会在消息选项卡中返回有关错误的更多详细信息，而不仅仅是错误号本身。

(无列名)	(无列名)	(无列名)	(无列名)	(无列名)	(无列名)
遇到以零作除数错误。	8134	16	1	3	NULL

ERROR_PROCEDURE()函数不能返回过程名，因为错误是在 ad-hoc 查询中生成的。

转换函数

数据类型转换可以通过 CAST()和 CONVERT()函数来实现。大多数情况下，这两个函数是重叠的，它们反映了 SQL 语言的演化历史。这两个函数的功能相似，不过它们的语法不同。虽然并非所有类型的值都能转变为其他数据类型，但总的来说，任何可以转换的值都可以用简单的函数实现转换。

CAST()函数

CAST()函数的参数是一个表达式，它包括用 AS 关键字分隔的源值和目标数据类型。以下例子用于将文本字符串'123'转换为整型:

```
SELECT CAST('123' AS int)
```

返回值是整型值 123。如果试图将一个代表小数的字符串转换为整型值，又会出现什么情况呢？

```
SELECT CAST('123.4' AS int)
```

CAST()函数和 CONVERT()函数都不能执行四舍五入或截断操作。由于 123.4 不能用 int 数据类型来表示，所以对这个函数调用将产生一个错误:

Server: Msg 245, **Level** 16, State 1, Line 1

Syntax error converting the **varchar** value

'123.4' **to** a **column of** data type **int**.

在将 varchar 值'123.4' 转换成数据类型 int 时失败。

要返回一个合法的数值，就必须使用能处理这个值的数据类型。对于这个例子，存在多个可用的数据类型。如果通过 CAST()函数将这个值转换为 decimal 类型，需要首先定义 decimal 值的精度与小数位数。在本例中，精度与小数位数分别为 9 与 2。精度是总的数字位数，包括小数点左边和右边位数的总和。而小数位数是小数点右边的位数。这表示本例能够支持的最大的整数值是 9999999，而最小的小数是 0.01。

```
SELECT CAST('123.4' AS decimal(9,2))
```

decimal 数据类型在结果网格中将显示有效小数位:123.40

精度和小数位数的默认值分别是 18 与 0。如果在 decimal 类型中不提供这两个值，SQL Server 将截断数字的小数部分，而不会产生错误。

```
SELECT CAST('123.4' AS decimal)
```

结果是一个整数值：123

在表的数据中转换数据类型是很简单的。下面的例子使用 Product 表，首先执行如下查询：

```
SELECT ProductNumber, ProductLine, ProductModelID
FROM Production.Product
WHERE ProductSubcategoryID < 4
```

假定产品经理已经创建了一个系统，用于唯一地标识生产出来的每辆自行车，以便跟踪其型号、类型和类别。他决定合并产品号、产品生产线标识符、产品型号标识符和一个顺序号，为生产出来的每辆自行车创建一个唯一的序列号。在这个过程的第一步，他要求提供包括除顺序号之外的所有属性的所有可能产品的根标识符。

如果使用下面的表达式，就不能得到希望的结果，如图 6-2 所示。

```
SELECT ProductNumber
       + '-'
       + ProductLine
       + '-'
       + ProductModelID AS BikeSerialNum
FROM Production.Product
WHERE ProductSubcategoryID < 4
```

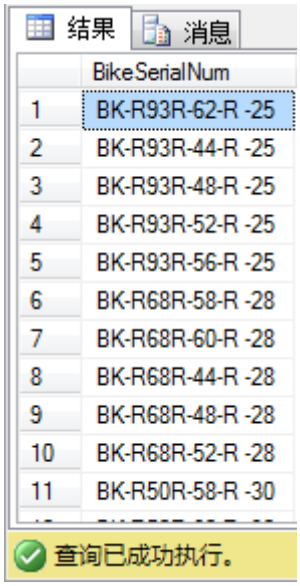
```
消息245，级别16，状态1，第1 行
在将nvarchar 值'BK-R93R-62-R -' 转换成数据类型int 时失败。
```

我们没有得到希望的结果，而得到了有点奇怪的错误消息：请把 nvarchar 值转换为 int。因为之前我们没有要求进行任何转换，所以这个错误很奇怪。这个查询的问题在于我们试图利用第一个连接符来连接字符值 ProductNumber，利用第二个连接符连接另一个字符值 ProductLine，最后连接的是 ProductModelID 字符值(它是一个整数)。

查询引擎会把连接符当成一个数学运算符，而不是一个字符。不管结果是什么，都需要更正这个表达式，以确保使用正确的数据类型。以下表达式执行了必要的类型转换，返回如图 6-3 所示的结果：

```
SELECT ProductNumber
       + '-'
       + ProductLine
       + '-'
       + CAST(ProductModelID AS char(4)) AS BikeSerialNum
FROM Production.Product
WHERE ProductSubcategoryID < 4
```

如果把整型值转换为字符类型就不会增加多余的空格了。查询引擎将把这些值用加号和连接符组合在一起，进行字符串连接运算，而不是和前面的数值进行加法或者减法运算了。



CONVERT()函数

对于简单类型转换，CONVERT()函数和 CAST()函数的功能相同，只是语法不同。CAST()函数一般更容易使用，其功能也更简单。CONVERT()函数的优点是可以格式化日期和数值，它需要两个参数：第 1 个是目标数据类型，第 2 个是源数据。以下的两个例子和上一节的例子类似：

```
SELECT CONVERT(int, '123')
SELECT CONVERT(decimal(9,2), '123.4')
```

CONVERT()函数还具有一些改进的功能，它可以返回经过格式化的字符串值，且可以把日期值格式化成很多形式。有 28 种预定义的符合各种国际和特殊要求的日期与时间输出格式。下表列出了这些日期格式。

如果 *expression* 为 date 或 time 数据类型，则 *style* 可以为下表中显示的值之一。其他值作为 0 进行处理。SQL Server 使用科威特算法来支持阿拉伯样式的日期格式。

yy(1)	yyyy	标准	输入/输出 (3)
-	0 或 100 (1, 2)	默认	mon dd yyyy hh:miAM（或 PM）
1	101	美国	mm/dd/yyyy
2	102	ANSI	yy.mm.dd
3	103	英国/法国	dd/mm/yyyy
4	104	德国	dd.mm.yy
5	105	意大利	dd-mm-yy
6	106 (1)	-	dd mon yy
7	107 (1)	-	mon dd, yy
8	108	-	hh:mi:ss
-	9 或 109 (1, 2)	默认设置 + 毫秒	mon dd yyyy hh:mi:ss:mmmAM（或 PM）
10	110	美国	mm-dd-yy
11	111	日本	yy/mm/dd
12	112	ISO	yymmdd yyyymmdd
-	13 或 113 (1, 2)	欧洲默认设置 + 毫秒	dd mon yyyy hh:mi:ss:mmm(24h)
14	114	-	hh:mi:ss:mmm(24h)
-	20 或 120 (2)	ODBC 规范	yyyy-mm-dd hh:mi:ss(24h)
-	21 或 121 (2)	ODBC 规范（带毫秒）	yyyy-mm-dd hh:mi:ss.mmm(24h)
-	126 (4)	ISO8601	yyyy-mm-ddThh:mi:ss.mmm（无空格）
-	127(6, 7)	带时区 Z 的 ISO8601。	yyyy-mm-ddThh:mi:ss.mmmZ（无空格）
-	130 (1, 2)	回历 (5)	dd mon yyyy hh:mi:ss:mmmAM
-	131 (2)	回历 (5)	dd/mm/yy hh:mi:ss:mmmAM

1. 这些样式值将返回不确定的结果。包括所有 (yy)（不带世纪数位）样式和一部分 (yyyy)（带世纪数位）样式。
2. 默认值（*style* 0 或 100、9 或 109、13 或 113、20 或 120 以及 21 或 121）始

终返回世纪数位 (yyyy)。

3. 转换为 datetime 时输入；转换为字符数据时输出。
4. 为用于 XML 而设计。对于从 datetime 或 smalldatetime 到字符数据的转换，其输出格式如上一个表所述。
5. 回历是有多种变体的日历系统。SQL Server 使用科威特算法。
   a) 默认情况下，SQL Server 基于截止年份 2049 年来解释两位数的年份。换言之，就是将两位数的年份 49 解释为 2049，将两位数的年份 50 解释为 1950。许多客户端应用程序（如基于自动化对象的应用程序）都使用截止年份 2030 年。SQL Server 提供了“两位数年份截止”配置选项，可通过此选项更改 SQL Server 使用的截止年份，从而对日期进行一致处理。建议您指定四位数年份。
6. 仅支持从字符数据转换为 datetime 或 smalldatetime。仅表示日期或时间成分的字符数据转换为 datetime 或 smalldatetime 数据类型时，未指定的时间成分设置为 00:00:00.000，未指定的日期成分设置为 1900-01-01。
7. 使用可选的时间区域指示符 (Z) 更便于将具有时区信息的 XML datetime 值映射到没有时区的 SQL Server datetime 值。Z 是时区 UTC-0 的指示符。其他时区则以 + 或 - 方向的 HH:MM 偏移量来指示。例如：2006-12-12T23:45:12-08:00。

从 smalldatetime 转换为字符数据时，包含秒或毫秒的样式将在这些位置上显示零。使用相应的 char 或 varchar 数据类型长度从 datetime 或 smalldatetime 值转换时，可截断不需要的日期部分。

从样式包含时间的字符数据转换为 datetimeoffset 时，将在结果末尾追加时区偏移量。

这个函数的第三个参数是可选的，该参数用于接收格式代码整型值。表中的例子用于对 DateTime 数据类型进行转换。在转换 SmallDateTime 数据类型时，格式不变，但一些元素会显示为 0，因为该数据类型不支持毫秒。以下的脚本例子将输出格式化的日期：

```
SELECT 'Default Date:' + CONVERT(Varchar(50), GETDATE(), 100)
```

Default Date: Apr 25 2005 1:05PM

```
SELECT 'US Date:' + CONVERT(Varchar(50), GETDATE(), 101)
```

US Date: 04/25/2005

```
SELECT 'ANSI Date:' + CONVERT(Varchar(50), GETDATE(), 103)
```

ANSI Date: 2005.04.25

```
SELECT 'UK/French Date:' +CONVERT (Varchar(50), GETDATE(), 103)
```

UK/French Date: 25/04/2OO5

```
SELECT 'German Date:' + CONVERT(Varchar(50), GETDATE(), 104)
```

German Date: 25.04.2005

格式代码 0，1 和 2 也可用于数字类型，它们对小数与千位分隔符格式产生影响。而不同的数据类型所受的影响是不一样的。一般来说，使用格式代码 0(或者不指定这个参数的值)，将返回该数据类型最惯用的格式。使用 1 或者 2 通常显示更为详细或者更精确的值。以下例子使用格式代码 0：

```
DECLARE @Num Money
SET @Num = 1234.56
SELECT CONVERT(varchar(50), @Num, 0)
```

返回结果如下:

1234.56

使用值 1 则返回如下结果:

1,234.56

使用值 2 则返回如下结果:

1234.5600

以下例子和上例相同，但是使用 Float 类型:

```
DECLARE @Num float
SET @Num = 1234.56
SELECT CONVERT(varchar(50), @Num, 2)
```

使用值 0 不会改变所提供的格式，但是使用值 1 或 2 将返回以科学计数法表示的数字，后者使用了 15 位小数：

1.23456000000000e+003

STR()函数

这是一个将数字转换为字符串的快捷函数。这个函数有 3 个参数：数值、总长度和小数位数。如果数字的整数位数和小数位数(要加上小数点占用的一个字符)的总和小于总长度，对结果中左边的字符将用空格填充。在下面第 1 个例子中，包括小数点在内一共是 5 个字符。结果显示在网格中，显然左边的空格被填充了。这个调用指定，总长度为 8 个字符，小数位为 4 位：

```
SELECT STR(123.4, 8, 4)
```

结果值的右边以 0 填充：123.4000。

下面给函数传递了一个 10 字符的值，并指定结果包含 8 个字符，有 4 个小数位：

```
SELECT STR(123.456789, 8, 4)
```

只有将这个结果截断才能符合要求。STR()函数对最后一位进行四舍五入：123.4568。现在，如果为函数传递数字 1，并指定结果包含 6 个字符，有 4 个小数位，STR()函数将用 0 补足右边的空位：

```
SELECT STR(1, 6, 4)
```

1.0000

然而，如果指定的总长度大于整数位数、小数点和小数位数之和，结果值的左边将用空格补齐：

```
SELECT STR(1, 6, 4)
```

1.0000

```
SELECT STR(1, 12, 4)
```

---------- 1.0000

游标函数与变量

游标可以处理多行数据，在过程循环中一次访问一行。和基于集合的高效操作相比，这个功能对系统资源的消耗更大。可以用一个函数和两个全局变量来管理游标操作。

CURSOR_STATUS()函数

这个函数返回一个整型值，表示传递给这个函数的游标类型变量的状态。有很多不同类型的游标会影响这个函数的操作。为简单起见，下表列出了这个函数的常见返回值。

返 回 值	说 明
1	游标包含一行或多行(动态游标包含 0 行或者多行)
0	游标不包含行

-1	游标已关闭
-2	游标未分配
-3	游标不存在

@@CURSOR_ROWS 全局变量

这个变量是一个整型值，表示在当前连接中打开的游标中的行数。根据游标类型，这个值也能不代表结果集中的实际行数。

@@FETCH_STATUS 全局变量

这个变量是一个标记，用于表示当前游标指针的状态。这个变量主要用来判断某行是否存在，以及在执行了 FETCH NEXT 语句后，是否已执行到结果集的尾部。打开游标时，@@FETCH_STATUS 变量值为-1。一旦把第一个值放在游标中，@@FETCH_STATUS 变量值就变成 0。当不再把更多的行放在游标中时，该变量的值将变回-1。

日期函数

这些函数可以操作 DateTime 与 SmallDateTime 类型的值。有些函数可用于解析日期值的日期与时间部分，有些函数可用于比较、操纵日期/时间值。日期数据类型的区别如下表所示。

数据类型	输出
time	12:35:29. 1234567
date	2007-05-08
smalldatetime	2007-05-08 12:35:00
datetime	2007-05-08 12:35:29.123
datetime2	2007-05-08 12:35:29. 1234567
datetimeoffset	2007-05-08 12:35:29.1234567 +12:15

DATEADD()函数

DATEADD()函数用于在日期/时间值上加上日期单位间隔。比如，要得到 2007 年 4 月 29 日起 90 天后的日期，可以使用下列语句：

```
SELECT DATEADD(DAY, 90, '4-29-2007')
```

结果：2007-07-28 00:00:00.000

可以把下表的值作为时间间隔参数传递给 DATEADD()函数。

datepart	缩写
year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
weekday	dw, w
hour	hh

minute	mi, n
second	ss, s
millisecond	ms
microsecond	mcs
nanosecond	ns

在下面列出的例子中，我们使用和上一个例子一样的日期，并且在这些例子中还包含了时间数据。每个操作的结果将显示在查询的下一行中。

18 年后：

```
SELECT DATEADD(YEAR, 18, '4-29-1988 10:30 AM')
```

2006-04-29 10:30:00.000

18 年前：

```
SELECT DATEADD(YY, -18, '4-29-1988 10:30 AM')
```

1970-04-29 10:30:00.000

9000 秒后：

```
SELECT DATEADD(SECOND, 9000, '4-29-1988 10:30 AM')
```

1988-04-29 13:00:00.000

9000000 毫秒前:

```
SELECT DATEADD(MS, -9000000, '4-29-1988 10:30 AM')
```

1988-04-29 08:00:00.000

可以将 CONVERT()函数和 DATEADD()函数组合在一起，来对 1989 年 9 月 8 日 9 个月前的日期值进行格式化。

```
SELECT CONVERT(varchar(20), DATEADD(M, -9, '9-8-1989'), 101)
```

12/08/1988

这将返回一个可变长度的字符值，比前面例子结果中的默认日期更易容易理解。这是一个函数嵌套调用，DATEADD()函数的返回值(一个 DateTime 类型的值)被作为值参数传递给 CONVERT()函数。

DATEDIFF()函数

DATEADD()和 DATEDIFF()函数可以看作一对表兄弟，有点像乘法与除法。在等式的两端有 4 个元素：起始日期、时间间隔(datepart)、差值和最终日期。如果已知其中的三个值，就可以求出第 4 个值。如果在 DATEADD()函数中使用起始日期、一个整型值和一个时间间隔，就可返回与起始日期相关的最终日期值。如果提供了起始日期、时间间隔和最终日期，DATEDIFF()函数就可以返回差值。

为了说明这一点，我们选择任意两个日期与一个时间间隔作为参数。这个函数将以所提供的时间间隔为单位返回两个日期之间的差值。要知道 1989 年 9 月 8 日和 1991 年 10 月 17 日之间差了几个月，可编写如下查询代码：

```
SELECT DATEDIFF(MONTH, '9-8-1989', '10-17-1991')
```

结果是 25 个月。如果以日期为单位呢?

```
SELECT DATEDIFF(DAY, '9-8-1989', '10-17-1991')
```

结果是 769 天。

1996 年 7 月 2 日和 1997 年 8 月 4 日之间差几个星期?

```
SELECT DATEDIFF(WEEK, '7-2-1996', '8-4-1997')
```

57 星期。甚至可以算出自己的年龄是多少秒：

```
DECLARE @MyBirthDate datetime
SET @MyBirthDate = '7-16-1962'
SELECT DATEDIFF(SS, @MyBirthDate, GETDATE())
```

结果显示有些人已经活了 15 亿秒了!

可以将列名作为参数，把这个函数用在查询中。首先建立一个简单的表，其中包含一些人的姓名和生日：

```
SELECT c.FirstName
    ,c.LastName
    ,e.BirthDate
    ,DATEDIFF(YEAR, e.BirthDate, GETDATE()) AS ApproximateAge
FROM HumanResources.Employee as e inner join
    Person.Contact as c on e.ContactID = c.ContactID
order by c.LastName
```

下图显示了结果：

结果 消息

	FirstName	LastName	BirthDate	ApproximateAge
1	Syed	Abbas	1965-02-11 00:00:00.000	45
2	Kim	Abercrombie	1957-01-14 00:00:00.000	53
3	Hazem	Abolrous	1967-11-27 00:00:00.000	43
4	Pilar	Ackerman	1962-10-11 00:00:00.000	48
5	Jay	Adams	1966-03-14 00:00:00.000	44
6	François	Ajenstat	1965-06-17 00:00:00.000	45
7	Amy	Alberts	1947-10-22 00:00:00.000	63
8	Greg	Alderson	1960-11-18 00:00:00.000	50
9	Sean	Alexander	1966-04-07 00:00:00.000	44
10	Gary	Altman	1961-03-21 00:00:00.000	49
11	Nancy	Anderson	1978-12-21 00:00:00.000	32

初看起来结果是对的，但存在的问题是年龄值没有精确到日。比如，根据表中的数据，Nancy 的生日是 12 月 21 日，他今年将庆祝第 32 个生日(这个查询在 2010 年 8 月运行)。如果依据上述计算结果来确定他的年龄何时变化，就应在一月份的某天给他发生日卡片，这比实际日期提前了 11 个月。

除非用更小的时间单位来计算这些日期的差，否则结果只在雇员实际生日的一年以内是精确的。以下例子将用差值除以一年(包括闰年)的天数，并将结果值转换为 int 类型，进行取整运算，而不是四舍五入。

```
SELECT c.FirstName
    ,c.LastName
    ,e.BirthDate
    ,DATEDIFF(YEAR, e.BirthDate, GETDATE()) AS ApproximateAge
     ,CONVERT(int, DATEDIFF(DAY, e.BirthDate, GETDATE())/365) AS Age
FROM HumanResources.Employee as e inner join
    Person.Contact as c on e.ContactID = c.ContactID
order by c.LastName
```

比较这次的结果和上一个例子的结果，看看有什么不同。

结果 消息

	FirstName	LastName	BirthDate	ApproximateAge	Age
1	Syed	Abbas	1965-02-11 00:00:00.000	45	45
2	Kim	Abercrombie	1957-01-14 00:00:00.000	53	53
3	Hazem	Abolrous	1967-11-27 00:00:00.000	43	42
4	Pilar	Ackerman	1962-10-11 00:00:00.000	48	47
5	Jay	Adams	1966-03-14 00:00:00.000	44	44
6	François	Ajenstat	1965-06-17 00:00:00.000	45	45
7	Amy	Alberts	1947-10-22 00:00:00.000	63	62
8	Greg	Alderson	1960-11-18 00:00:00.000	50	49
9	Sean	Alexander	1966-04-07 00:00:00.000	44	44
10	Gary	Altman	1961-03-21 00:00:00.000	49	49
11	Nancy	Anderson	1978-12-21 00:00:00.000	32	31

可以看到，Nancy 是 31 岁，其他雇员的年龄也精确到了天。表中的 BirthDate 列存储雇员的生日，并以午夜(00:00:00AM)为界，这是一天中的第一秒。GETDATE()函数返回当前的时间与日期。当前两个日期相差约 8 小时(写这段文字时是上午 8 点)。如果希望这个计算更精确，就需要在当前日期的午夜把 GETDATE()函数的结果转换为 datetime 类型。

DATEPART()与 DATENAME()函数

这两个函数用于返回 datetime 或者 shortdatetime 值的日期部分。DATEPART()函数返回一个整型值；DATENAME()函数返回一个包含描述性文字的字符串。比如，将日期 4-29-1988 传递给 DATEPART()函数，如指定返回月份值，则返回数字 4：

```
SELECT DATEPART(MONTH, '4-29-1988')
```

而使用相同的参数，DATENAME()函数返回 04（这取决于你的机器的本地语言，如果是英文版，那么将返回 April）：

```
SELECT DATENAME(MONTH, '4-29-1988')
```

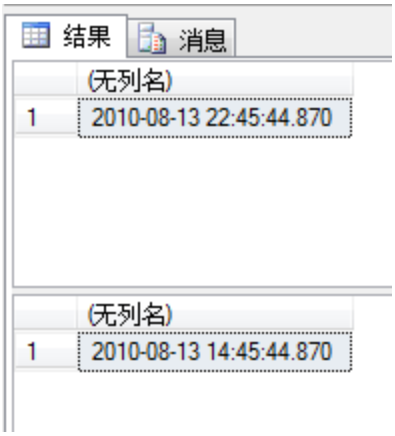
这两个函数都接收和 DATEADD()函数一样的时间间隔参数常量。

GETDATE()与 GETUTCDATE()函数

这两个函数都用于返回 datetime 类型的当前日期与时间。GETUTCDATE()函数使用服务器上的时区设置来求出 UTC 时间，这和格林威治标准时间或飞行员所说的"祖鲁时"(Zulu Time)是一样的。两个函数都能精确到 3.33 毫秒。

```
SELECT GETDATE()
SELECT GETUTCDATE()
```

执行这两个函数，都将返回未经格式化的结果，见下图：



我在北京，和 UTC 时间相差 8 个小时，和标准时间相差 9 个小时。可以使用如下 DATEDIFF() 函数来验证这个时间差值：

```
SELECT DATEDIFF(HOUR, GETDATE(), GETUTCDATE())
```

SYSDATETIME()和 SYSUTCDATETIME()函数

这两个 SQL Server 2008 函数等价于 GETDATE()和 GETUTCDATE()函数，但不是返回 datetime 数据类型的结果，而是返回 SQL Server 2008 新的 datetime2 数据类型的结果，该数据类型可以精确到 100 纳秒，当然这取决于服务器安装的硬件。

```
SELECT SYSDATETIME()
SELECT SYSUTCDATETIME()
```

DAY()、MONTH()和 YEAR()函数

这三个函数分别返回以整数表示的 datetime 或者 smalldatetime 类型值的日、月、年。它们的用途很广泛，如可以创建独特的个性化日期格式。假设需要创建一个自定义的日期值作为字符串，通过将这三个函数的输出结果转换成字符类型，然后进行连接操作，就可以对输出结果以任何形式进行组合了：

```
SELECT 'Year: ' + CONVERT(varchar(4), YEAR(GETDATE()))
    + ', Month: ' + CONVERT(varchar(2), MONTH(GETDATE()))
    + ', Day: ' + CONVERT(varchar(2), DAY(GETDATE()))
```

这个脚本生成下列结果：

Year:2008, Month:2, Day:20

下一节将讨论字符串操纵函数，并使用相似的技术来构建一个紧凑的定制时间戳。

字符串操纵函数

字符串函数可以解析、替换、操纵字符型值。在处理原始字符数据时，最大的挑战之一是如何可靠地提取出有意义的信息。有很多字符串解析函数可用于标识和解析子字符串(一个大字符型值的一部分)。我们一直在做这种事，在我们阅读文件、发票或者书面材料时，就会本能地标识、分离出有意义的信息片段。这个过程的自动化非常困难，即使是处理不太复杂的文本，也很困难。这些函数包含几乎所有必需的工具，而挑战在于如何找出最简单、最高效的方法。

ASCII()、CHAR()、UNICODE()和 NCHAR()函数

这四个函数是相似的，它们都可以在字符和字符的标准数字表示之间转换。美国标准信息交换码(American Standard Code for Information Interchange，ASCII)标准字符集包含 128 个字母、数字和标点符号。这个字符集是 IBM PC 体系结构的基础，虽然有些字符现在看来已经很古老了，但还是被保留了下来，且仍是现代计算机技术的核心。如果在计算机上使用英语，则键盘上的每个字符都是用 ASCII 码表示的。这对说英语(至少以英语打字)的计算机用户来说是有利的，但是其他人又该怎么办呢?

在计算机的发展过程中， ASCII 字符集发布没多长时间便过时了。人们很快将它扩展成为 256 个字符的 ANSI 字符集，一个字符用一个字节来保存。这个扩展的字符列表满足了许多其他用户的需求，可以支持主要的欧洲语言字符，不过仍是美国标准(由美国国家标准学会持有)，仍建立在最初的英语字符集的基础上。为了支持所有可印刷的语言，人们制订了 Unicode 标准，它支持多种语言特定的字符集。每个 Unicode 字符需要 2 个字节的存储空间，是 ASCII 与 ANSI 字符的两倍。但是使用 2 个字就可以表示超过 65 000 个不同的字符，完全能够支持东欧和亚洲字符。SQL Server 同时支持 ASCII 与 Unicode 两种标准。

ASCII()和 CHAR()是两个基于 ASCII 的函数，这两个函数可将计算机上应用的每个字符表示为数字。要确定代表一个字符的数字是什么，就应给 ASCII()函数传送只包含一个字符的字符串，如下：

```
SELECT ASCII('A')
```

结果是 65。

如要将一个已知数字转换为字符，又该怎么办？使用 CHAR()函数即可：

```
SELECT CHAR(65)
```

结果是字母 A。

要得到完整的 ASCII 字符值列表，可以对一个临时表填充从 0 到 127 的数字，然后调用 CHAR()函数返回相应的字符。为了节省空间，我们对以下这个脚本进行了删节，但包含整个结果集，并以多栏格式给出。

```
-- 创建一个临时表来保存ASCII码:
Create Table #ASCIIVals (ASCIIValue smallint)
-- 插入数字0 - 127 到临时表中:
declare @Number int
set @Number = 0
while(@Number < 128)
    begin
        Insert Into #ASCIIVals (ASCIIValue) Select @Number
        set @Number = @Number + 1
    end
-- 查询所有的整型数字与其对应的ASCII码:
SELECT ASCIIValue, CHAR(ASCIIValue) AS Character FROM #ASCIIVals
drop table #ASCIIVals
```

表 6-12 是以多栏网格重新格式化的结果集。需要注意的是这里将不可印刷的控制字符以方括号表示。由于许多因素限制，如所安装的字体或语言不同，下表的显示可能会有稍许差异。

0		16	┼	32		48	0	64	@	80	P	96	`	112	p
1		17	◄	33	!	49	1	65	A	81	Q	97	a	113	q
2	┐	18	↕	34	"	50	2	66	B	82	R	98	b	114	r
3	└	19	‼	35	#	51	3	67	C	83	S	99	c	115	s
4	┘	20	¶	36	$	52	4	68	D	84	T	100	d	116	t
5	│	21	┴	37	%	53	5	69	E	85	U	101	e	117	u
6	─	22	┬	38	&	54	6	70	F	86	V	102	f	118	v
7	•	23	┤	39	'	55	7	71	G	87	W	103	g	119	w
8	◘	24	↑	40	(	56	8	72	H	88	X	104	h	120	x
9		25	├	41	)	57	9	73	I	89	Y	105	i	121	y
10		26	→	42	*	58	:	74	J	90	Z	106	j	122	z
11	♂	27	←	43	+	59	;	75	K	91	[	107	k	123	{
12	♀	28		44	,	60	<	76	L	92	\	108	l	124	|
13		29		45	-	61	=	77	M	93	]	109	m	125	}
14	♫	30		46	.	62	>	78	N	94	^	110	n	126	~
15	☼	31		47	/	63	?	79	O	95	_	111	o	127	[]

UNICODE()函数是 ASCII()的 Unicode 等价函数，NCHAR()函数和 CHAR()函数的功能相同，只不过 NCHAR()是用于 Unicode 字符的。SQL Server 的 nchar 与 nvarchar 类型能存储任何 Unicode 字符，可以和这两个函数一起使用。对于特别大的值，ntext 类型和 nvarchar(max)类型也支持 Unicode 字符。

要返回扩展字符编码集中的字符，可以将字符编码传递给 NCHAR()函数：

```
SELECT NCHAR(220)
```

返回字母ü。

```
SELECT NCHAR(233)
```

返回带重音符号的小写 e：é。

```
SELECT NCHAR(241)
```

返回西班牙语的"enya"，或者带有发音符号的 n：ñ。

当然，ASCII 标准也支持所有的欧洲字符，所以使用 CHAR()函数也可以返回这些扩展字符。如果对 256~65536 之间的值使用 CHAR()函数，返回值就很有趣了。例如，下面的查询返回希腊字符Ω：

```
SELECT NCHAR(433)
```

下面的查询返回西里尔字母 Ya(Я)。

```
SELECT NCHAR(1071)
```

CHARINDEX()和 PATINDEX()函数

CHARINDEX()是原始的 SQL 函数，用于寻找在一个字符串中某子字符串第一次出现的位置。如函数名所示，这个函数返回一个整型值，表示某子字符串的第一个字符在整个字符串中的位置索引。以下脚本用于在字符串 Washington 中寻找子字符串 sh 的出现位置:

```
SELECT CHARINDEX('sh', 'Washington')
```

返回的结果是 3，表明 s 是字符串 Washington 中的第 3 个字符。这说明 CHARINDEX 函数匹配字符的索引是从 1 开始的。如果没有匹配到任何结果，函数将返回 0。在这个例子中使用两个字符作为子字符串并没有特别意义，但是如果字符串包含多个 s 字符，就有意义了。

PATINDEX()函数和 CHARINDEXO 函数类似，它执行相同的操作，但方法稍许不同，该函数增加了对通配符(即 Like 运算符中使用的字符)的支持。顾名思义，它将返回一个字符模式

的索引。这个函数也可以和 ntext、nchar(max)和 nvarchar(max)等大字符类型一起使用。注意，如果和这些大字符类型一起使用，PATINDEX()函数将返回 bigint 类型的值，而不是 int 类型的值。以下是一个例子：

```
SELECT PATINDEX('%M_rs%', 'The stars near Mars are far from ours')
```

注意，如果想找到一个字符串，在所比较的字符串的前后各有 0 个或者多个字符，则两个百分符都是必须的。下划线表明这个位置上的字符不必匹配，它可以是任意字符。

和使用相同字符串的 CHARINDEX()函数作一下比较：

```
SELECT CHARINDEX('Mars', 'The stars near Mars are far from ours')
```

这两个函数都返回索引值 16。请注意这些函数的执行过程。下一节将把这两个函数和 SUBSTRING()函数组合在一起，演示如何使用界定符解析字符串。

LEN()函数

LEN()函数用于返回一个代表字符串长度的整型值。这是一个简单、有用的函数，经常与其他函数一起使用，来应用业务规则。以下例子将月份和日期转换为字符类型，然后测试它们的长度。如果月份日期只有一个字符，就填充字符 0，然后组合成一个 8 字符的美国格式的日期字符串(MMDDYYYY)。

```
DECLARE @MonthChar varchar(2), @DayChar varchar(2), @DateOut char(8)
SET @MonthChar = CAST(MONTH(GETDATE()) AS varchar(2))
SET @DayChar = CAST(DAY(GETDATE()) AS varchar(2))
-- Make sure month and day are two char long:
IF LEN(@MonthChar) = 1
  SET @MonthChar = '0' + @MonthChar
IF LEN(@DayChar) = 1
  SET @DayChar = '0' + @DayChar
-- Build date string:
SET @DateOut = @MonthChar + @DayChar + CAST(YEAR(GETDATE()) AS char(4))
SELECT @DateOut AS OutputDate
```

这个脚本将返回代表日期的 8 个字符:

08152010

LEFT()和 RIGHT()函数

LEFT()与 RIGHT()函数是相似的，它们都返回一定长度的子字符串。这两个函数的区别是，它们返回的分别是字符串的不同部分。LEFT()函数返回字符串最左边的字符，顺序从左数到右。RIGHT()函数正好相反，它从最右边的字符开始，以从右到左的顺序返回特定数量的字符。看一看使用这两个函数返回"GeorgeWashington"这个字符串的子字符串的例子。

如果使用 LEFT()函数返回一个 5 字符的子字符串，则函数先定位最左边的字符，向右数 5 个字符，然后返回这个子字符串，如下所示。

```
DECLARE @FullName varchar(25)
SET @FullName = 'George Washington'
SELECT LEFT(@FullName, 5)
```

结果为：Georg

如果使用 RIGHT()函数返回一个 5 字符的子字符串，则函数先定位最右边的字符，向左数 5 个字符，然后返回这个子字符串，如下所示。

```
DECLARE @FullName varchar(25)
SET @FullName = 'George Washington'
SELECT RIGHT (@FullName, 5)
```

结果为：ngton

要想返回字符串中有意义的部分，这两个函数都不是特别有用。如果想返回全名中的姓氏或者名字，该怎么办？这需要多做一点工作。如果能确定每个姓名中空格的位置，就可以使用 LEFT()函数在全名中读取名字。在这种情况下，可以使用 CHARINDEX()或者 PATINDEX()函数来定位空格，然后使用 LEFT()函数返回空格前的字符。下面是第一个用过程方法编写的例子，它将处理过程分解成以下步骤：

```
DECLARE @FullName varchar(25), @SpaceIndex tinyint
SET @FullName = 'George Washington'
-- Get index of the delimiting space:
SET @SpaceIndex = CHARINDEX(' ' , @FullName)
-- Return all characters to the left of the space:
SELECT LEFT(@FullName, @SpaceIndex - 1)
```

结果为：George

如果不想在结果中包含空格，就需要从@SpaceIndex 值中减去 1，这样结果中就只有名字了。

SUBSTRING()函数

SUBSTRING()函数能够从字符串的一个位置开始，往右数若干字符，返回一个特定长度的子字符串。和 LEFT()函数不同之处是，该函数可以指定从哪个位置开始计数，这样就可以在字符串的任何位置摘取子字符串了。这个函数需要三个参数：要解析的字符串、起始位置索引、要返回的子字符串长度。如果要返回到所输入字符串尾部的所有字符，可以使用比所需长度更大的长度值。SUBSTRING()函数将返回最大可能长度的字符数，而不会将多出的长度以空格填充。

只要指定字符串最左边的字符(1)为起始索引，就可以用 SUBSTRING()函数替代 LEFT()函数。

继续上一节的例子。可以设置起始位置与长度，返回姓名字符串中间的值。在这个例子中，从位置 4 开始，返回一个 6 字符的子字符串"rge Wa"。

```
DECLARE @FullName varchar(25)
SET @FullName = 'George Washington'
SELECT SUBSTRING(@FullName, 4, 6)
```

现在将上述各函数组合在一起，即可从名字+空格+姓氏格式的全名字符串中解析出名字和姓氏。使用先前的逻辑，通过函数嵌套来减少脚本的行数，并去掉@SpaceIndex 变量。下面用 SUBSTRING()函数替代 LEFT()函数：

```
DECLARE @FullName varchar(25)
SET @FullName = 'George Washington'
-- Return first name:
SELECT SUBSTRING(@FullName, 1, CHARINDEX(' ', @FullName) - 1)
```

类似的逻辑可以用于解析姓氏，但是必须将起始位置更改为空格后的那个字符。如果空格在第 7 个位置上，那么姓氏将从第 8 个位置开始。这就意味着起始位置是 CHARINDEX()的返回结果加上 1。

```
DECLARE @FullName varchar(25)
SET @FullName = 'George Washington'
--Return last name:
SELECT SUBSTRING(@FullName, CHARINDEX(' ', @FullName) + 1,
  LEN(@FullName))
```

把上述步骤组合在一起，就可以运行下面的查询，从全名变量中提取出名字和姓氏：

```
DECLARE @FullName varchar(25)
SET @FullName = 'George Washington'
-- Return first name:
SELECT SUBSTRING(@FullName, 1, CHARINDEX(' ',@FullName) - 1) AS
FirstName,
      SUBSTRING(@FullName, CHARINDEX(' ',@FullName) + 1, LEN(@FullName))
      AS LastName
```

结果为：

FirstName	LastName
George	Washington

传递给 SUBSTRING()函数的值是空格所在位置加上 1，并将该值作为起始位置，这将是姓氏的第 1 个字母。由于不可能总是知道名字的长度，所以将 LEN()函数的结果作为子字符串长度参数传递进来，当 SUBSTRING()函数到达这个位置时，就到达了字符串的末尾，这样就可以将字符串中从空格后面开始的所有字符都包含进来了。

为了举例方便，先创建并填充一个临时表:

```
CREATE TABLE #MyNames (FullName varchar(50))
GO
INSERT INTO #MyNames (FullName) SELECT 'Fred Flintstone'
INSERT INTO #MyNames (FullName) SELECT 'Wilma Flintstone'
INSERT INTO #MyNames (FullName) SELECT 'Barney Rubble'
INSERT INTO #MyNames (FullName) SELECT 'Betty Rubble'
INSERT INTO #MyNames (FullName) SELECT 'George Jetson'
INSERT INTO #MyNames (FullName) SELECT 'Jane Jetson'
go
--drop table #MyNames
```

下面执行一个使用函数调用来解析名字和姓氏值的单行查询表达式。这里对@FullName 变量的引用被表中的 FullName 列所替代:

```
SELECT
    SUBSTRING(FullName, 1, CHARINDEX(' ', FullName) - 1) AS FirstName
    ,SUBSTRING(FullName, CHARINDEX(' ', FullName) + 1, LEN(FullName)) AS
LastName
FROM #MyNames
```

在下图所示的结果中，显示了两个不同的列，分别是名字和姓氏。

FirstName	LastName
Fred	Flintstone
Wilma	Flintstone
Barney	Rubble
Betty	Rubble
George	Jetson
Jane	Jetson

LOWER()和 UPPER()函数

这两个函数很容易理解，它们用于将字符串中所有字符分别都转换为小写和大写，这在比较用户输入或者存储用于比较的字符串时是非常有用的。字符串比较通常是区分大小写的，这取决于 SQL Server 安装时的设置。如果和其他的字符串操纵函数一起使用，就可以将字符串转换为合适的大小写，以便存储或显示。以下例子说明混合大小写的名字，假设名字中的第 2 个大写子字符串前只包含一个空格，但在特殊情况下也有一些名字是没有空格的。这个例子很容易通过扩展来处理包含其他类型的混合大小写名字(如以 MC 开头的名字，带连接号的名字等)。

```
DECLARE @LastName varchar(25), @SpaceIndex tinyint
SET @LastName = 'mc donald'             -- Test value
-- Find space in name:
SET @SpaceIndex = CHARINDEX(' ' , @LastName)
IF @SpaceIndex > 0                      -- Space: Capitalize first &
substring
    SELECT UPPER(LEFT(@LastName, 1))
  + LOWER(SUBSTRING(@LastName, 2, @SpaceIndex - 1))
  + UPPER(SUBSTRING(@LastName, @SpaceIndex + 1, 1))
  + LOWER(SUBSTRING(@LastName, @SpaceIndex + 2, LEN(@LastName)))
ELSE                                    -- No space: Cap only first char.
    SELECT UPPER(LEFT(@LastName, 1))
  + LOWER(SUBSTRING(@LastName, 2, LEN(@LastName)))
```

这个脚本将返回 MC Donald。还可以对这个例子进行扩展，以处理姓氏包含撇号的情况。在这个例子的业务规则中，空格是不考虑的。如果找到了撇号，就将后面的字符全部转为大写。请注意如果要在脚本中测试撇号，就必须输入两次撇号(' ')，以表明这是一个文字，而不是一对单引号。姓氏中只存储一个撇号。

```
DECLARE @LastName varchar(25), @SpaceIndex tinyint, @AposIndex tinyint
SET @LastName = 'o''malley'             -- Test value
-- Find space in name:
SET @SpaceIndex = CHARINDEX(' ', @LastName)
-- Find literal ' in name:
SET @AposIndex = CHARINDEX('''', @LastName)
IF @SpaceIndex > 0                      -- Space: Capitalize first &
substring
    SELECT UPPER(LEFT(@LastName, 1))
  + LOWER(SUBSTRING(@LastName, 2, @SpaceIndex - 1))
  + UPPER(SUBSTRING(@LastName, @SpaceIndex + 1, 1))
```

```
  + LOWER(SUBSTRING(@LastName, @SpaceIndex + 2, LEN(@LastName)))
ELSE IF @AposIndex > 0                    -- Apostrophe: Cap first & substring
    SELECT UPPER(LEFT(@LastName, 1))
  + LOWER(SUBSTRING(@LastName, 2, @AposIndex - 1))
  + UPPER(SUBSTRING(@LastName, @AposIndex + 1, 1))
  + LOWER(SUBSTRING(@LastName, @AposIndex + 2, LEN(@LastName)))
ELSE                                      -- Nospace: Cap only first char.
    SELECT UPPER(LEFT(@LastName, 1))
  + LOWER(SUBSTRING(@LastName, 2, LEN(@LastName)))
```

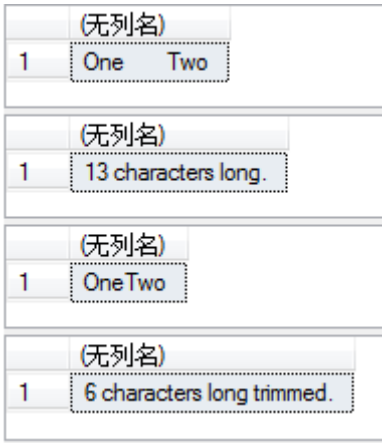
这个脚本返回 O'Malley。

LTRIM()和 RTRIM()函数

这两个函数分别返回将字符串的左边和右边的空白修剪掉之后的字符串:

```
DECLARE @Value1 char(10), @Value2 char(10)
SET @Value1 = 'One'
SET @Value2 = 'Two'
SELECT @Value1 + @Value2
SELECT CONVERT(varchar(5), LEN(@Value1 + @Value2))
+ ' characters long. '
SELECT RTRIM(@Value1) + RTRIM(@Value2)
SELECT CONVERT(varchar(5), LEN(RTRIM(@Value1) + RTRIM(@Value2)))
      + ' characters long trimmed. '
```

结果如下：



REPLACE()函数

REPLACE()函数可以把字符串中的某个字符或某个子字符串替换为另一个字符或者子字符串，该函数可以用于全局查找和替换工具中。

```
DECLARE @Phrase varchar(1000)
SET @Phrase = 'I aint gunna use poorgrammar when commenting script and
I aint gunna complain about it. '
SELECT REPLACE(@Phrase, 'aint', 'am not')
```

REPLICATE()和 SPACE()函数

在需要将一些字符重复填充进一个字符串时，这两个函数是非常有用的。这里也使用 SUBSTRING()例子中的临时表为每个名字填满 20 个字符，然后将 20 减去各个字符串的长度，以便将正确的值传递给 REPLICATE()函数:

```
SELECT FullName + REPLICATE('*', 20 - LEN(FullName))
FROM #MyNames
```

结果是每个名字后面都填满了星号，各个名字的总长度都是 20 个字符:

Fred Flintstorle*****

Wilrna Flintstone****

Barney Rubble*******

Betty Rubble********

George Jetson********

Jane Jetson**********

SPACE()函数与上述函数类似，区别在于该函数使用空格进行填充。它返回一个由空格组成的字符串，空格的个数由参数定义。

```
SELECT FullName + SPACE(20 - LEN(FullName))
FROM  #MyNames
```

如果返回"#MyNames 表不存在"的错误，只需再次运行本文前面"SUBSTRING()函数"一节的 CREATE TABLE 脚本即可。

REVERSE()函数

顾名思义，这个函数用于将字符串中的字符颠倒过来。这在处理连接列表中的单个字符值时将会被用到。

```
SELECT REVERSE('The stars near Mars are far from ours. ')
```

结果为：.sruo morf raf era sraM raen srats ehT

STUFF()函数

这个函数可将字符串中的一部分替换为另一个字符串。它本质上是将一个字符串以特定的长度插入另一个字符串中的特定位置上。这对于源值与目的值的长度不一样的字符串替换是很有用的。下列代码用于将字符串中的价格替换为 109.95:

Please submit your payment for 99.95 immediately.

价格值是从第 32 个字符开始的，有 5 个字符长。在这个位置上插入的子字符串有多长并不重要，只需要知道需要删除多少个字符就可以了。

```
SELECT STUFF('Please submit your payment for 99.95 immediately. ',
32, 5, '109.95')
```

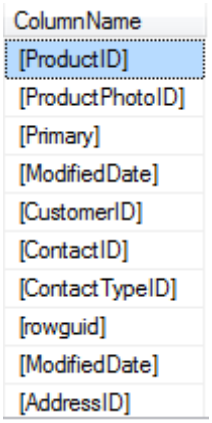
结果为：Please submit your payment for 109.95 immediately.

QUOTENAME()函数

这个函数和 SQL Server 对象名组合使用，以将结果传递给表达式。它只用于给输入的字符串加一对方括号，并返回新形成的字符串。如果参数包含保留的分隔符或者封装字符(比如引号或括号)，这个函数将修改字符串，以便 SQL Server 能将结果字符串中的这类字符当成文本字符。如下面的例子所示，查询的结果如图 6-10 所示。

```
SELECT QUOTENAME(COLUMN_NAME) AS ColumnName
```

```
FROM INFORMATION_SCHEMA. COLUMNS
```



数学函数

下表中列出的函数用于执行多种普通与特殊的数学运算，可以执行代数、三角、统计、估算与财政运算等运算。

函 数	说 明
ABS()	返回一个数的绝对值
ACOS()	计算一个角的反余弦值，以弧度表示
ASIN()	计算一个角的反正弦值，以弧度表示
ATAN()	计算一个角的反正切值，以弧度表示
ATN2()	计算两个值的反正切，以弧度表示
CEILING()	返回大于或等于一个数的最小整数
COS()	计算一个角的正弦值，以弧度表示
COT()	计算一个角的余切值，以弧度表示
DEGREES()	将一个角从弧度转换为角度
EXP()	指数运算
FLOOR()	返回小于或等于一个数的最大整数
LOG()	计算以 2 为底的自然对数
LOG10()	计算以 10 为底的自然对数
PI()	返回以浮点数表示的圆周率
POWER()	幂运算
RADIANS()	将一个角从角度转换为弧度
RAND()	返回以随机数算法算出的一个小数， 可以接收一个可选的种子值
ROUND()	对一个小数进行四舍五入运算， 使其具备特定的精度
SIGN()	根据参数是正还是负，返回 - 1 或者 1
SIN()	计算一个角的正弦值，以弧度表示
SQRT()	返回一个数的平方根
SQUARE()	返回一个数的平方
TAN()	计算一个角正切的值，以弧度表示

元数据函数

这是一些工具函数，它们返回 SQL Server 配置细节、服务器与数据库设置细节的信息，包括一组用于返回不同对象的属性状态的通用以及专用函数，这些函数把对 Master 数据库中系统表以及用户数据库的查询封装在函数中。建议读者使用这些函数以及其他的系统函数，而不是自己创建对系统表的查询，以防今后 SQL Server 版本对模式进行更改。

排列函数

这些函数被用于以与结果集顺序无关的特定顺序，枚举已排序的或排在前面的结果集。

ROW_NUMBER()函数

ROW_NUMBER()函数根据作为参数传递给这个函数的 ORDER BY 子句的值，返回一个不断递增的整数值。如果 ROW_NUMBER 的 ORDER BY 的值和结果集中的顺序相匹配，返回值将是递增的，以升序排列。如果 ROW_NUMBER 的 ORDER BY 子句的值和结果集中的顺序不同，这些值将不会按顺序列出，但它们表示 ROW_NUMBER 函数的 ORDER BY 子句的顺序。如下面的例子和结果所示：

```
SELECT ProductCategoryID
    ,Name
    ,ROW_NUMBER() OVER (ORDER BY Name) AS RowNum
FROM Production.ProductCategory
ORDER BY Name
```

由于 ROW_NUMBER()调用中的 ORDERBY 子句和查询结果的顺序匹配，所以对这些结果按顺序列出，如下图所示：

ProductCategoryID	Name	RowNum
4	Accessories	1
1	Bikes	2
3	Clothing	3
2	Components	4

不过，在函数调用中使用另一个 ORDER BY 子句时，这些值就是无序的了。

```
SELECT ProductCategoryID
    ,Name
    ,ROW_NUMBER() OVER (ORDER BY Name) AS RowNum
FROM Production.ProductCategory
ORDER BY ProductCategoryID
```

这是了解如何使用 ORDER BY 子句对结果进行排序的有效方法。如下图所示：

ProductCategoryID	Name	RowNum
1	Bikes	2
2	Components	4
3	Clothing	3
4	Accessories	1

RANK()与 DENSE_RANK()函数

这两个函数与 ROW_NUMBER()函数类似，因为它们都返回一个基于 ORDER BY 子句的值。不过这些值不一定永远是唯一的。排列值对于所提供的 ORDER BY 子句中的重复结果而言也

是重复的，而且唯一性是仅仅基于 ORDER BY 列表中的唯一值的。这些函数用不同的方法来处理重复的值。RANK()函数保留列表中行的位置序号，对于每个重复的值，该函数会跳过下面与其相邻的值，于是就可以将下一个不重复的值保留在正确的位置上。

其行为类似于短跑比赛中的并列成绩。例如刘翔与 Dayron Robles（古巴）在 110 栏的比赛中都跑出了 12’92 的成绩，那他们就是并列第一，而其后的一名选手将会获得第三名的成绩。

```
SELECT ProductID
    ,Name
    ,ListPrice
    ,RANK() OVER (ORDER BY ListPrice DESC) AS [Rank]
FROM Production.Product
ORDER BY [Rank]
```

注意在下图的结果列表中，重复的价格值所对应的结果是相同的，而每个连接之后的值都被跳过了。比如，产品"Road-150 Red, 52"和"Road-150 Red, 56"都排在第 1，而接下来的行"Mountain-100 Silver,38"就排在第 6 了。

ProductID	Name	ListPrice	Rank
749	Road-150 Red, 62	3578.27	1
750	Road-150 Red, 44	3578.27	1
751	Road-150 Red, 48	3578.27	1
752	Road-150 Red, 52	3578.27	1
753	Road-150 Red, 56	3578.27	1
771	Mountain-100 Silver, 38	3399.99	6
772	Mountain-100 Silver, 42	3399.99	6
773	Mountain-100 Silver, 44	3399.99	6
774	Mountain-100 Silver, 48	3399.99	6
775	Mountain-100 Black, 38	3374.99	10
776	Mountain-100 Black, 42	3374.99	10
777	Mountain-100 Black, 44	3374.99	10
778	Mountain-100 Black, 48	3374.99	10
789	Road-250 Red, 44	2443.35	14

DENSE_RANK()函数的工作方式与 RANK()函数相同，不过它不会跳过每个连接后的值，这样就不会有值被跳过了，但是在连接处排列序号位置将会丢失。

```
SELECT ProductID
    ,Name
    ,ListPrice
    ,DENSE_RANK() OVER (ORDER BY ListPrice DESC) AS [Rank]
FROM Production.Product
ORDER BY [Rank]
```

下图的结果重复了排列值，但是不会跳过列中的任何数字。

ProductID	Name	ListPrice	Rank
749	Road-150 Red, 62	3578.27	1
750	Road-150 Red, 44	3578.27	1
751	Road-150 Red, 48	3578.27	1
752	Road-150 Red, 52	3578.27	1
753	Road-150 Red, 56	3578.27	1
771	Mountain-100 Silver, 38	3399.99	2
772	Mountain-100 Silver, 42	3399.99	2
773	Mountain-100 Silver, 44	3399.99	2
774	Mountain-100 Silver, 48	3399.99	2
775	Mountain-100 Black, 38	3374.99	3
776	Mountain-100 Black, 42	3374.99	3
777	Mountain-100 Black, 44	3374.99	3
778	Mountain-100 Black, 48	3374.99	3
789	Road-250 Red, 44	2443.35	4

NTILE(n)函数

这个函数也用于对结果进行排列，并返回一个整型的排列值，但是它不会对结果以唯一的排列顺序进行枚举，而是将结果切分为有限数量的排列组。比如，一个表有 10 000 行，使用 1000 为参数值调用 NTILE()函数，即 NTILE(1000)，并将结果分成以 10 为单位的 1000 个组，每个组赋予相同的排列值。和本节讨论的其他排列函数一样，NTILE()函数也支持 OVER(ORDER BY…)语法。下面的例子根据产品价格，按照从高到低的顺序把 Product 表分为 50 组产品：

```
SELECT ProductID
    ,Name
    ,ListPrice
    ,NTILE(50) OVER (ORDER BY ListPrice DESC) AS GroupedProducts
FROM Production.Product
ORDER BY GroupedProducts
```

结果为：

ProductID	Name	ListPrice	GroupedProducts
749	Road-150 Red, 62	3578.27	1
750	Road-150 Red, 44	3578.27	1
751	Road-150 Red, 48	3578.27	1
752	Road-150 Red, 52	3578.27	1
753	Road-150 Red, 56	3578.27	1
771	Mountain-100 Silver, 38	3399.99	1
772	Mountain-100 Silver, 42	3399.99	1
773	Mountain-100 Silver, 44	3399.99	1
774	Mountain-100 Silver, 48	3399.99	1
775	Mountain-100 Black, 38	3374.99	1
776	Mountain-100 Black, 42	3374.99	1
777	Mountain-100 Black, 44	3374.99	2
778	Mountain-100 Black, 48	3374.99	2
789	Road-250 Red, 44	2443.35	2
790	Road-250 Red, 48	2443.35	2
791	Road-250 Red, 52	2443.35	2
792	Road-250 Red, 58	2443.35	2

安全函数

与安全相关的函数返回 SQL Server 用户的角色成员和权限信息。这类函数也包括一组管理事件与跟踪的函数。下表显示了这些函数：

函 数	说 明
fn_trace_geteventinfo()	为指定的跟踪 ID 返回一个填充事件信息的表类型值
fn_trace_getfilterinfo()	为指定的跟踪 ID 返回一个填充与过滤器有关的信息的表类型值
fn_trace_getinfo()	为指定的跟踪 ID 返回一个填充跟踪信息的表类型值
fn_trace_getable()	为指定的跟踪 ID 返回一个填充文件信息的表类型值
HAS_DBACCESS()	返回一个表明当前用户是否有访问指定数据库权限的标志
IS_MEMBER()	返回一个表明当前用户是 Windows 组用户还是 SQL Server 用户的标志
IS_SRVROLEMEMBER()	返回一个表明当前用户是否是数据库服务器角色成员的标志
SUSER_SID()	返回指定用户的登录名的安全 ID，或者(如果参数被忽略)返回当前用户的安全 ID。返回指定用户的用户 ID，或者(如果参数被忽略的话)返回当前用户的用户 ID
SUSER_SNAME()	返回指定安全 ID 的登录名。如果不提供任何安全 ID，则返回当前用户的登录名
USER_ID()	返回指定用户名的用户 ID，或者(如果参数被忽略的话)返回当前用户的用户 ID
USER_NAME()	返回指定用户 ID 的用户名

系统函数与系统变量

本节讨论具有多种用途的工具函数，包括值比较、值类型测试等功能。这个类别的函数也包罗了其他函数：

函 数	说 明

APP_NAME()	返回与当前连接相关联的应用程序的名字
COALESCE()	从以逗号分隔的表达式列表中返回第一个非空值
COLLATIONPROPERTY()	返回一个特定字符集排序规则的特定属性的值。这些属性包括 CodePage、LCID、ComparisonStyle
CURRENT_TIMESTAMP()	返回当前日期与时间。和 GETDATE()函数是同义的。这个函数的存在只是为了与 ANSI-SQL 兼容
C1UJRRENT_USER()	返回当前用户的名字。与 USER_NAME()函数相同
DATALENGTH()	返回存储或处理一个值所需的字节数。对于 ANSI 字符串类型，这个函数返回的值与 LEN()函数相同，但对于其他数据类型而言就可能不一定相同了
fn_helpcollations()	返回一个填充有由当前 SQLSewer 版本支持的字符集排序规则的表类型值
fn_servershareddrives()	返回一个填充有服务器共享的驱动列表的表类型值
fn_virtualfilestats()	返回一个填充有包括日志文件在内数据库文件的 I/O 状态的表类型值
FORMATMESSAGE()	从 sysmessages 表中为指定的信息代码和以逗号分隔的参数列表返回错误信息
GETANSINULL()	根据 ANSLNULL_DFLT_ON 与 ANSLNULL_DFLT_OFF 数据库设置返回数据库的可空性设置
HOST_ID()	返回当前会话的工作站 ID
HOST_NAME()	返回当前会话的工作站名
IDENT_CURRENT()	返回最后一个为指定的表生成的标识(ID)值。与会话、范围无关
IDENT_INCR()	返回最后一次创建的标识(ID)列中定义的增量值
IDENT_SEED()	返回最后一次创建的标识(ID)列中定义的种子值
IDENTITY()	用在 SELECT…INTO 语句中，在一个列中插入自动生成的标识值
ISDATE()	返回一个表明指定的值是否可被转换为日期值的标志
ISNULL()	判断指定的值是否是空值，然后返回一个事先提供的替代值
ISNUMERIC()	返回一个表明指定的值是否可被转换为数字值的标志
NEWID()	返回一个新生成的 UniqueIdentifier 类型的值。这是一个 128 位的整型、全球唯一的值，通常以字母或数字十六进制来表示(如 89DE6247・C2E242DB-8CE8・A787E505D7EA)。这个类型经常被用作复制的和半连接系统中的主键.
NULLIF()	两个特定的参数的值如果是相同的，则返回 NULL
PARSENAME()	返回一个具有 4 部分对象名的特定部分
PERMISSIONS()	返回一个整型值，该值是一个表示当前用户在指定的数据库对象上权限或者权限组合的位映像
ROWCOUNT_BIG()	与@@RowCount 变量一样，这个函数返回被最后一条语句修改或返回的行数量。返回值类型是 bigint
SCOPE_IDENTITY()	与@@IDENTIY 变量一样，这个函数返回限制在当前会话与范围内的最后一次生成的标识值
SERVERPROPERTY()	返回一个表示服务器属性状态的标记。属性包括 Collation、

	Edition 、 EngineEdition 、 InstanceName 、 IsClustered 、 IsFullTextInstalled、IsIntegrated- SecurityOnly、IsSingleUser、IsSyncWithBackup、LicenseTYpe、MachineName、NumLicenses、ProcessID、ProductLevel、ProductVersion、ServerName
SESSION_USER	返回当前用户名。调用本函数不需要括号
SESSIONPROPERTY()	返回表示一个会话属性状态的标记。属性包括:ANSL_NULLS，ANSI_PADDING ， ANSL_WARNINGS ， ARITHABORT ，CONCAT_NULL_ YIELDS_NULL ， NUMERIC_ROUNDABORT ，QUOTED_IDENTIFIER
STATS_DATE()	返回指定的索引统计信息最后一次被更新的时间
SYSTEM_USER	返回当前用户名。调用本函数不需要括号
USER_NAME()	为一个指定的用户 ID 返回用户名。如果没有提供 ID 号则返回当前的数据库用户

COALESCE()函数

COALESCE()函数是非常有用的，它返回其参数中第一个非空表达式。它能够节省颇多 IF 或者 CASE 分支逻辑。以下例子用产品数据填充一个表，每个产品最多有 3 种价格:

```
CREATE TABLE #ProductPrices (ProductName varchar(25), SuperSalePrice
Money NULL, SalePrice Money NULL, ListPrice Money NULL)
GO
INSERT INTO #ProductPrices VALUES('Standard Widget', NULL, NULL, 15.95)
INSERT INTO #ProductPrices VALUES('Economy Widget', NULL, 9.95, 12.95)
INSERT INTO #ProductPrices VALUES('Deluxe Widget', 19.95, 20.95, 22.95)
INSERT INTO #ProductPrices VALUES('Super Deluxe Widget', 29.45, 32.45,
38.95)
INSERT INTO #ProductPrices VALUES('Executive Widget', NULL, 45.95, 54.95)
GO
```

所有的产品都有定价，有些有销售价，有些还有促销价。一项产品的当前价格是所有已有价格的最低价，或者在读取每个价格列时以列出顺序读到的第一个非空值:

```
SELECT ProductName, COALESCE(SuperSalePrice, SalePrice, ListPrice) AS
CurrentPrice
FROM #ProductPrices
```

这个方法比使用多行分支与判断逻辑要简洁得多，而结果也是同样简单，如下图所示:

ProductName	CurrentPrice
Standard Widget	15.95
Economy Widget	9.95
Deluxe Widget	19.95
Super Deluxe Widget	29.45
Executive Widget	45.95

DATALENGTH()函数

DATALENGTH()函数返回一个用于对值进行管理的字节数，这有助于揭示不同数据类型间的一些有趣差别。当把 varchar 类型传递给 DATALENGTH()和 LEN()函数时，它们将返回相同的值:

```
DECLARE @Value varchar(20)
SET @Value = 'abc'
SELECT DATALENGTH(@Value)
SELECT LEN(@Value)
```

这些语句的返回值都为 3。因为 varchar 类型使用了 3 个单字节字符来存储三个字符的值。然而，如果使用 nVarchar 类型来管理相同长度的值，就要占用多一倍的字节:

```
DECLARE @Value nvarchar(20)
SET @Value = 'abc'
SELECT DATALENGTH(@Value)
SELECT LEN(@Value)
```

DATALENGTH()函数返回值为 6，因为每个使用 Unicode 字符集的字符都要占用 2 个字节。LEN()函数返回值为 3，因为这个函数返回字符数，不是字节数。以下是一个有趣的测试：要存储一个值为 2 的整型变量，要占用多少个字节？而如果要存储一个值为 20 亿的整型变量，又将占用多少个字节呢？试一下：

```
DECLARE @Value1 int, @Value2 int
SET @Value1 = 2
SET @Value2 = 2000000000
SELECT DATALENGTH(@Value1)
SELECT LEN(@Value1)
SELECT DATALENGTH(@Value2)
SELECT LEN(@Value2)
```

在这两种情况下，DATALENGTH()函数都返回 4。因为 int 类型不论值是多少，总是使用 4 个字节。LEN()函数本质上将整型值当成已转换成字符型的数据来处理，所以，在这个例子中，它分别返回 1 和 10，即值的位数。

在下表中的全局系统变量都将返回 int 类型的值。这些变量可用于存储过程和其他实现定制业务逻辑的编程对象。

变 量	说 明
@@ERROR	当前会话最后一次发生的错误代码
@@IDENTITY	当前会话最后一次生成的标识值
@@ROWCOUNT	当前会话中最后一次返回结果集的执行操作所返回的行数
@@TRANCOUNT	当前会话中活动的事务数。这是在执行相关的 COMMIT TRANSACTION 或者 ABORT TRANSACTION 语句之前嵌套的多个 BEGIN TRANSACTION 语句的结果

系统统计变量

下表描述了用于确定数据库系统使用信息与环境信息的管理工具：

变 量	说 明
@@CONNECTIONS	返回打开连接的次数
@@CPU_BUSY	从上次启动服务器开始，SQL Server 一共工作的毫秒数
@@IDLE	从上次启动服务器开始，SQL Server 一共空闲的毫秒数
@@IO_BUSY	从上次启动服务器开始，SQL Server 一共处理 I/O 的毫秒数
@@PACK_RECEIVED	从上次启动服务器开始，SQL Server 一共收到的网络数据包数
@@PACK_SENT	从上次启动服务器开始，SQL Server 一共发送的网络数据包数

@@PACKET_ERRORS	从上次启动服务器开始，SQL Server 一共收到的网络数据包错误数
@@TIMETICKS	每个时钟滴答有多少毫秒
@@TOTAL_ERRORS	从上次启动服务器开始，SQL Server 一共收到的磁盘 I/O 错误数
@@TOTAL_READ	从上次启动服务器开始，SQL Server 一共进行的物理磁盘读取次数
@@TOTAL_WRITE	从上次启动服务器开始，SQL Server 一共进行的物理磁盘写入次数

小结

函数用于实现业务逻辑，并且能够将编程功能带入查询中。许多有用而且强大的函数是 T-SQL 的标准功能。和面向过程、面向对象语言中的函数一样，SQL 函数也将程序功能封装到一个简单的可重用的包中，这就减少了查询设计人员的很多工作。由于 Transact-SQL 是面向任务的语言，而不是过程语言。虽然函数可以进行过程编程，可以在查询中构建颇为复杂的逻辑，但是 SQL 语言的优势在于让设计人员表达出设计意图，而不是完成一项任务的确切步骤与方法。只要使用方法正确，这些步骤和方法都可以由函数来实现。

在 T-SQL 中，参数用于将值传递给函数，大多数函数的返回结果是一个标量，或者说单一值。函数分为确定性函数与非确定性函数。在使用相同的参数时，确定性函数总是返回相同的值，而非确定性函数的返回值则与其他资源有关，所以 SQL Server 必须显式地执行这种函数。因此，在定制的 SQL 编程对象中，对非确定性函数的使用是有限制的。

SQL 函数执行种类繁多的重要任务，包括数学运算、比较、日期解析与操纵、高级字符串操纵等。