13

Пользовательские функции

Пользовательские функции принадлежат к числу наиболее привлекательных объектов SQL Server. Возможность применения пользовательских функций (User Defined Function — UDF) появилась больше пяти лет тому назад, но до сих пор они остаются одними из самых недостаточно используемых и недооцененных объектов SQL Server. Эти объекты произвели потрясающее впечатление на специалистов по базам данных сразу после их внедрения корпорацией Microsoft в версии SQL Server 2000, но со времени появления инфраструктуры .NET пользовательские функции приобрели еще бо́льшие возможности. А с точки зрения читателя настоящей книги, который ознакомился со всеми предыдущими главами, одна из наиболее замечательных особенностей пользовательских функций состоит в том, что ему уже известно почти все, что требуется для создания этих функций. Фактически пользовательские функции чрезвычайно напоминают хранимые процедуры и отличаются от последних только тем, что обладают некоторыми дополнительными характеристиками и возможностями, которые подчеркивают их особенности и обеспечивают применение во многих сложных ситуациях.

В настоящей главе приведено вводное описание пользовательских функций, рассматриваются различные типы пользовательских функций, подчеркивается их отличие от хранимых процедур, а также, безусловно, приводится описание тех ситуаций, в которых может возникнуть необходимость ими воспользоваться. Наконец, даны краткие сведения о том, как можно использовать инфраструктуру .NET для расширения области применения пользовательских функций.

Общее описание пользовательских функций

Пользовательские функции во многом напоминают хранимые процедуры и представляют упорядоченное множество операторов T-SQL, которые заранее оптимизированы, откомпилированы и могут быть вызваны для выполнения работы в виде единого модуля. Основное различие между пользовательскими функциями и хранимыми процедурами состоит в том, как в них осуществляется возврат полученных результатов. А в связи с тем, что для обеспечения предусмотренного в них способа возврата значений в пользовательских функциях должны осуществляться немного другие действия, к их синтаксической структуре предъявляются более жесткие требования по сравнению с хранимыми процедурами.

Для полноты изложения автор обязан подчеркнуть, что между пользовательскими функциями и хранимыми процедурами есть не только сходство, но и различие. Прежде всего, пользовательские функции, безусловно, не могут рассматриваться как замена для хранимых процедур; они представляют собой всего лишь еще один способ организации кода, позволяющий получить дополнительные возможности.

Хранимые процедуры позволяют передавать входные параметры и получать сформированные в них значения в виде возвращаемых выходных параметров. Безусловно, с помощью хранимой процедуры также можно предусмотреть возврат значения в точку вызова, но в действительности это значение предназначено для использования в качестве индикатора успешного или неудачного завершения, а не в качестве возвращаемых данных. Кроме того, хотя с помощью хранимой процедуры можно обеспечить возврат результирующих наборов, фактически эти результирующие наборы нельзя применять для дальнейшей работы с ними в каком-то запросе без предварительной вставки в какую-то таблицу (обычно во временную таблицу).

С другой стороны, при использовании пользовательских функций допускается передавать входные параметры, но выходные параметры в них не предусмотрены. Но отказ от использования выходных параметров компенсируется введением в действие гораздо более надежно формируемого возвращаемого значения. Возвращаемое значение может быть скалярным, как и в случае применения системных переменных, но особенно привлекательным свойством пользовательских функций является то, что тип данных возвращаемого значения не ограничивается только целочисленным типом, как при использовании хранимых процедур. Значения, возвращаемые пользовательской функцией, могут относиться почти к любому типу данных SQL Server (дополнительная информация по этой теме приведена в следующем разделе).

Но возможности формирования возвращаемых значений пользовательской функции не ограничиваются лишь скалярными значениями; допускается также использовать в качестве возвращаемых значений таблицы. Такая возможность является чрезвычайно удобной, и дополнительные сведения по этой теме будут приведены ниже в данной главе.

На этом основании можно отметить, что пользовательские функции подразделяются на два описанных ниже типа.

- Возвращающие скалярное значение.
- Возвращающие таблицу.

Рассмотрим общее определение синтаксиса оператора создания пользовательской функции:

```
CREATE FUNCTION [<schema name>.]<function name>
    ( [ <@parameter name> [AS] [<schema name>.]<scalar data type> [ =
<default value>]
      [ ,...n ] ] )
RETURNS {<scalar type>|TABLE [(<Table Definition>)]}
    [ WITH [ENCRYPTION]|[SCHEMABINDING]|
[ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ] | [EXECUTE AS {
CALLER|SELF|OWNER|<'user name'>} ]
]
[AS] { EXTERNAL NAME <external method> |
BEGIN
    [<function statements>]
    {RETURN <type as defined in RETURNS clause>|RETURN (<SELECT statement>)}
END }[;]
```

Очевидно, что синтаксис оператора `CREATE FUNCTION` является довольно сложным, поскольку возможность применения необязательных частей этой синтаксической структуры зависит от того, какие компоненты были выбраны в других частях оператора создания пользовательской функции. При этом многое зависит от того, возвращает ли функция данные скалярного типа или таблицу, а также создается ли функция, основанная на использовании операторов языка T-SQL, или формируется функциональная структура, в которой применяются средства CLR и .NET. Поэтому ниже различные варианты синтаксической структуры данного оператора рассматриваются отдельно.

Пользовательские функции, возвращающие скалярное значение

По-видимому, пользовательские функции, возвращающие скалярное значение, больше всего напоминают функции, обычно применяемые в программном обеспечении. Как и большинство собственных встроенных функций SQL Server (таких как `GETDATE()` или `USER()`), пользовательские функции такого типа возвращают в вызывающий их сценарий или процедуру скалярное значение.

Как было указано выше, одной из наиболее привлекательных особенностей пользовательских функций является то, что при работе с ними можно не ограничиваться применением в качестве возвращаемых значений данных целочисленного типа, поэтому возвращаемые значения могут относиться к любому допустимому типу данных SQL Server (включая определяемые пользователем типы данных!), кроме данных типа BLOB, курсоров и временны*х* отметок. Способ оформления кода в виде пользовательской функции является весьма привлекательным (даже если необходимо обеспечить лишь возврат целочисленного значения) по двум описанным ниже причинам.

- В хранимых процедурах возвращаемое значение предназначено для использования в качестве индикатора успеха или неудачи, причем в случае неудачного завершения возвращаемое значение предоставляет некоторую конкретную информацию о характере возникшего нарушения в работе, а в пользовательских функциях, напротив, возвращаемое значение служит исключительно в качестве осмысленного фрагмента данных.

- Функции могут вызываться на выполнение как непосредственно встроенные в запрос (например, могут входить в состав оператора SELECT), а хранимые процедуры не предоставляют такой возможности.

Рассмотрим пример создания простой пользовательской функции, который позволяет подчеркнуть такие особенности функций данного типа, благодаря которым они могут использоваться иначе по сравнению с хранимыми процедурами. Безусловно, в качестве иллюстрации можно было бы выбрать пользовательскую функцию, более простую по сравнению с рассматриваемой в данном примере, но она позволяет более наглядно показать различия между хранимыми процедурами и пользовательскими функциями.

По мнению автора, один из наиболее удобных способов использования функции состоит в том, что с ее помощью подготавливаются данные для ввода в поле типа datetime информации о том, что некоторое событие произошло в какой-то определенный день. Обычно при решении такой задачи возникает проблема, связанная с тем, что в поле типа datetime имеется конкретная информация о времени суток, в связи с наличием которой затрудняется сравнение хранимого значения со значением, содержащим только одну дату. Безусловно, с этой проблемой мы уже сталкивались в предыдущих главах, когда требовалось реализовать некоторые операции сравнения значений дат.

Вернемся к базе данных Accounting, которая была создана в одной из предыдущих глав. Предположим, что необходимо получить сведения обо всех заказах, полученных за сегодняшний день. Начнем с того, что внесем в список несколько заказов, в которых проставлена сегодняшняя дата. Для этого просто выберем известные нам идентификаторы заказчиков и служащих из соответствующих таблиц (если в таблицах с данными о заказчиках и служащих базы данных Accounting еще нет строк, то необходимо вставить для обеспечения доступа к ним несколько фиктивных строк). Сам автор для ввода нескольких строк собирается применить небольшой цикл:

```
USE Accounting
DECLARE @Counter  int
SET @Counter = 1
WHILE @Counter <= 10
BEGIN
   INSERT INTO Orders
      VALUES (1, DATEADD(mi,@Counter,GETDATE()), 1)
   SET @Counter = @Counter + 1
END
```

Итак, при выполнении этого сценария происходит вставка десяти строк, в каждой из которых содержится сегодняшняя дата, но строки, следующие друг за другом, отличаются по времени на одну минуту.

Отметим, что при вызове этого сценария непосредственно перед полуночью некоторые из строк могут перескочить на следующие сутки и замысел данного примера не будет раскрыт, поэтому будьте осторожны. Но для всех остальных читателей, кроме полуночников, этот пример будет очень наглядным.

Таким образом, мы можем приступить к выполнению простого сценария, позволяющего определить, какие заказы были введены сегодня. Для этой цели можно попытаться применить примерно такой оператор:

```
SELECT *
FROM Orders
WHERE OrderDate = GETDATE()
```

Но, к сожалению, этот запрос не возвратит ни одной строки. Это связано с тем, что функция GETDATE() возвращает не только дату, но и текущее время, вплоть до миллисекунды. Это означает, что вероятность получения каких-либо данных с помощью запроса, в котором используется функция GETDATE() в чистом виде, является очень низкой, даже если интересующее нас событие произошло в тот же день (чтобы операция сравнения завершилась успешно, должно было быть так, что сравниваемые события произошли в одну и ту же минуту, если используются данные о времени типа smalldatetime, и в течение одной миллисекунды, если используется полный формат datetime).

Обычно при таких обстоятельствах применяется решение, в котором предусматривается прямое и обратное преобразование даты в строку для удаления информации о времени, после чего выполняется операция сравнения.

Соответствующий оператор может выглядеть приблизительно так:

```
SELECT *
FROM Orders
WHERE CONVERT(varchar(12), OrderDate, 101) = CONVERT(varchar(12),
GETDATE(), 101)
```

На сей раз будут получены все строки, в которых столбец OrderDate содержит сегодняшнюю дату, независимо от того, в какое время дня был введен заказ. Но, к сожалению, этот код нельзя назвать наиболее удобным для чтения. К тому же он очень громоздкий. А если в программе приходится предусматривать подобные операции сравнения для целого ряда дат, то соответствующий сценарий приобретает действительно сложный вид.

Поэтому рассмотрим способ выполнения тех же действий, но с помощью простой пользовательской функции. Вначале необходимо создать саму функцию. Эта задача осуществляется с помощью оператора нового типа CREATE FUNCTION, а применяемый при этом синтаксис во многом напоминает синтаксис создания хранимой процедуры. Например, указанную функцию можно реализовать с помощью такого кода:

```
CREATE FUNCTION dbo.DayOnly(@Date datetime)
RETURNS varchar(12)
AS
BEGIN
   RETURN CONVERT(varchar(12), @Date, 101)
END
```

При использовании этой функции дата, возвращаемая функцией GETDATE(), передается в качестве параметра, задача преобразования даты реализуется в теле функции и осуществляется возврат усеченного значения даты.

Чтобы ознакомиться с действием этой функции, внесем соответствующие изменения в приведенный выше запрос:

```
SELECT *
FROM Orders
WHERE dbo.DayOnly(OrderDate) = dbo.DayOnly(GETDATE())
```

После выполнения этого запроса будет получен тот же результирующий набор, как и при использовании обычного запроса. Вполне очевидно, что даже в таком простом

примере, как этот, удобство кода для чтения значительно повышается, а сам вызов осуществляется в основном так же, как и в большинстве языков программирования, которые поддерживают функции. Тем не менее возникает один нюанс — приходится учитывать такое понятие, как схема. По некоторым причинам в СУБД SQL Server поиск объектов, соответствующих именам функции, происходит иначе по сравнению с другими объектами.

На основании приведенного примера можно также сделать вывод, что пользовательские функции предоставляют гораздо больше преимуществ, чем просто повышение удобства чтения. В эти функции можно встраивать запросы, после чего применять функции как метод инкапсуляции для подзапросов. Кроме того, в пользовательских функциях можно инкапсулировать код процедурной реализации почти любых алгоритмов, возвращающий дискретное значение, после чего непосредственно вводить такие функции в запрос.

Рассмотрим очень простой пример подзапроса. Версия этого подзапроса выглядит следующим образом:

```
USE pubs
SELECT Title,
   Price,
   (SELECT AVG(Price) FROM Titles) AS Average, Price - (SELECT AVG(Price)
FROM Titles)
     AS Difference
FROM Titles
WHERE Type='popular_comp'
```

Выполнение приведенного кода приводит к получению довольно простого набора данных:

```
Title                     Price         Average       Difference
------------------------  ------------  ------------  -------------
But Is It User Friendly?  22.9500       14.7662       8.1838
Secrets of Silicon Valley 20.0000       14.7662       5.2338
Net Etiquette             NULL          14.7662       NULL
(3 row(s) affected)
Warning: Null value is eliminated by an aggregate or other SET operation.
```

Предпримем еще одну попытку использования функций, но на этот раз инкапсулируем в виде функций обе операции — и операцию вычисления среднего, и операцию вычитания. Первая функция инкапсулирует операцию вычисления среднего, а вторая — операцию вычитания:

```
CREATE FUNCTION dbo.AveragePrice()
RETURNS money
WITH SCHEMABINDING
AS
BEGIN
   RETURN (SELECT AVG(Price) FROM dbo.Titles)
END
GO
CREATE FUNCTION dbo.PriceDifference(@Price money)
RETURNS money
AS
BEGIN
   RETURN @Price - dbo.AveragePrice()
END
```

Обратите внимание на то, что вполне допустимо вкладывать одну пользовательскую функцию в другую.

Следует отметить, что опция WITH SCHEMABINDING осуществляет применительно к функциям такие же действия, как и по отношению к представлениям, – если функция создается с использованием привязки к схеме, то любой объект, от которого зависит функция, не может быть модифицирован или уничтожен без предварительного удаления привязанной к схеме функции. В данном примере фактически привязка к схеме не требовалась, но автор хотел проиллюстрировать использование этой опции, а также подготовить данный пример для применения с той целью, с какой он потребуется немного позже в настоящей главе.

Теперь вызовем тот же запрос на выполнение и применим в нем новую функцию, а не прежнюю модель с подзапросом:

```
USE pubs
SELECT Title,
   Price,
   dbo.AveragePrice() AS Average,
   dbo.PriceDifference(Price) AS Difference
FROM Titles
WHERE Type='popular_comp'
```

В результате формируются те же результаты, что и при использовании подзапроса, но предупреждающее сообщение не появляется!

Следует отметить, что пользовательские функции не только способствуют повышению удобства чтения кода, но и предоставляют дополнительное преимущество, связанное с неоднократным использованием кода. Небольшие примеры, подобные приведенному выше, по-видимому, не позволяют показать, насколько важным является это преимущество, но по мере усложнения применяемых функций экономия трудозатрат разработчиков становится весьма значительной.

Пользовательские функции, которые возвращают таблицу

Возможности применения пользовательских функций в СУБД SQL Server не ограничиваются лишь возвратом с их помощью скалярных значений. Эти функции обеспечивают возврат гораздо более важных объектов — таблиц. Из этого следуют такие возможности, которые трудно представить себе сразу же, но отметим, что возвращаемая таблица в большинстве обстоятельств доступна для применения в основном с использованием таких же способов, как и любая другая таблица. Результаты, возвращаемые функцией, можно включать в состав операндов операции JOIN и даже применять к ним условия конструкций WHERE. Благодаря этому открываются действительно заманчивые перспективы.

Изменения, которые должны быть внесены в пользовательскую функцию для получения возможности использовать таблицу в качестве возвращаемого значения, не являются сложными, поскольку, что касается таких функций, таблица рассматривается наряду с любым другим типом данных SQL Server. Чтобы проиллюстрировать сказанное, вначале создадим относительно простую функцию:

```
USE pubs
GO
```

```
CREATE FUNCTION dbo.fnAuthorList()
RETURNS TABLE
AS
RETURN (SELECT au_id,
          au_lname + ', ' + au_fname AS au_name,
          address AS address1,
          city + ', ' + state + ' ' + zip AS address2
        FROM authors)
GO
```

В этой функции выполняется возврат таблицы, состоящей из строк, полученных с помощью оператора SELECT, а также несложное форматирование: конкатенация фамилии и имени, разделенных запятыми, а также конкатенация трех компонентов адреса для возврата значений столбца address2.

С этого момента созданная функция может использоваться по такому же принципу, как таблица, за единственным исключением — как было указано при описании скалярных функций, при обозначении имени функции необходимо использовать соглашение об именовании, касающееся двухкомпонентных имен:

```
SELECT *
FROM dbo.fnAuthorList()
```

Полученные результаты имеют довольно большой объем (табл. 13.1), поэтому приведены лишь начало и конец таблицы, но эти данные наглядно демонстрируют суть действий, выполняемых данной функцией.

Таблица 13.1. Результаты выборки данных с применением функции `fnAuthorList()`

Столбец au_id	Столбец au_name	Столбец address1	Столбец address2
172-32-1176	White, Johnson	10932 Bigge Rd.	Menlo Park, CA 94025
213-46-8915	Green, Marjorie	309 63rd St. #411	Oakland, CA 94618
238-95-7766	Carson, Cheryl	539 Darwin Ln.	Berkeley, CA 94705
...	...	...	...
...	...	...	...
893-72-1158	McBadden, Heather	301 Putnam	Vacaville, CA 95688
899-46-2035	Ringer, Anne	67 Seventh Av.	Salt Lake City, UT 84152
998-72-3567	Ringer, Albert	67 Seventh Av.	Salt Lake City, UT 84152

Но приведенный выше пример еще не позволяет судить обо всех возможностях пользовательских функций. Ведь таблицу, сформированную в этом примере, вполне можно было получить столь же просто (а в действительности даже проще) с помощью представления. Но если бы нам потребовалось параметризовать представление или предусмотреть возможность его применения для получения информации только о тех авторах, которые написали книги, распроданные по меньшей мере в определенном количестве, то задание стало бы гораздо сложнее. Безусловно, можно было бы решить указанные задачи, соединив полученную таблицу еще с одной или с двумя таблицами, но отметим, что и в этом случае применяемый код стал бы гораздо более громоздким и могла бы возникнуть необходимость включить в представление дополнительный столбец, без которого в других условиях можно было обойтись (столбец с данными о количестве проданных книг), а затем применить конструкцию WHERE.

В результате реализации указанного подхода был бы получен примерно такой сценарий создания представления:

```
-- Создание представления
CREATE VIEW vSalesCount
AS
   SELECT au.au_id,
      au.au_lname + ', ' + au.au_fname AS au_name,
      au.address,
      au.city + ', ' + au.state + ' ' + zip AS address2,
      SUM(s.qty) As SalesCount
   FROM authors au
   JOIN titleauthor ta
      ON au.au_id = ta.au_id
   JOIN sales s
      ON ta.title_id = s.title_id
   GROUP BY au.au_id,
      au.au_lname + ', ' + au.au_fname,
      au.address,
      au.city + ', ' + au.state + ' ' + zip
GO
```

Разумеется, этот сценарий позволяет решить поставленную задачу, но с учетом некоторых нюансов. Во-первых, невозможно предусмотреть применение параметров непосредственно в самом представлении, поэтому в запрос придется ввести конструкцию `WHERE`. Во-вторых, потребуется предоставить конкретный список выборки для исключения столбца `vSalesCount` (напомним, что задача состоит в том, чтобы показать авторов, количество проданных книг которых превысило указанное значение, но не обязательно фактические объемы сбыта книг этих авторов). Поэтому приходится применять для получения данных оператор

```
SELECT au_name, address, Address2 FROM vSalesCount
WHERE SalesCount > 25
```

Выполнение этого оператора приводит к получению результатов, которые выглядят так, как показано в табл. 13.2.

Таблица 13.2. Результаты выполнения запроса, в котором используется представление `vSalesCount`

Столбец `au_name`	Столбец `address`	Столбец `address2`
Green, Marjorie	309 63rd St. #411	Oakland, CA 94618
Carson, Cheryl	589 Darwin Ln.	Berkeley, CA 94705
O'Leary, Michael	22 Cleveland Av. #14	San Jose, CA 95128
Dull, Ann	3410 Blonde St.	Palo Alto, CA 94301
DeFrance, Michel	3 Balding Pl.	Gary, IN 46403
MacFeather, Stearns	44 Upland Hts.	Oakland, CA 94612
Panteley, Sylvia	1956 Arlington Pl.	Rockville, MD 20853
Hunter, Sheryl	3410 Blonde St.	Palo Alto, CA 94301
Ringer, Anne	67 Seventh Av.	Salt Lake City, UT 84152
Ringer, Albert	67 Seventh Av.	Salt Lake City, UT 84152

Но если вместо этого все необходимые операторы будут инкапсулированы в виде функции, то применяемый способ решения задачи значительно упростится:

```
USE pubs
GO
CREATE FUNCTION dbo.fnSalesCount(@SalesQty bigint)
RETURNS TABLE
AS
RETURN (SELECT au.au_id,
          au.au_lname + ', ' + au.au_fname AS au_name,
          au.address,
          au.city + ', ' + au.state + ' ' + zip AS Address2
       FROM authors au
       JOIN titleauthor ta
          ON au.au_id = ta.au_id
       JOIN sales s
          ON ta.title_id = s.title_id
       GROUP BY au.au_id,
          au.au_lname + ', ' + au.au_fname,
          au.address,
          au.city + ', ' + au.state + ' ' + zip
       HAVING SUM(qty) > @SalesQty
       )
GO
```

Таким образом, мы провели весьма неплохую предварительную подготовку, и, чтобы выполнить нужный нам запрос, достаточно вызвать функцию и передать ей параметр:

```
SELECT *
FROM dbo.fnSalesCount(25)
```

При этом будет получен точно такой же результирующий набор, но для этого не приходится применять конструкцию `WHERE`, исключать ненужные столбцы с помощью списка выборки, а также испытывать какие-либо другие затруднения. Кроме того, единожды созданную функцию можно вызывать на выполнение снова и снова, не прибегая каждый раз к формированию текста запроса с помощью надоевшего метода “вырезки и вставки”. К тому же следует отметить, что, даже несмотря на возможность достижения аналогичных результатов с помощью хранимой процедуры и оператора `EXEC`, таблицу, полученную с помощью хранимой процедуры, нельзя непосредственно соединить с другой таблицей.

Чтобы проиллюстрировать сказанное, немного дополним рассматриваемый пример. Предположим, что менеджер из отдела сбыта хочет иметь возможность формировать отчет со списком, в котором указан каждый автор и его издатель (издатели), если количество проданных книг этого автора превышает 25. Для получения данного отчета необходимо выполнить операцию соединения, но хранимая процедура не позволяет решить такую задачу непосредственно, поэтому нелегко сразу же найти способ выхода из этого затруднения. (Автор знает, как это сделать с помощью определенного процесса, состоящего из нескольких шагов, но этот процесс является довольно трудоемким.) А в случае использования приведенной выше функции никаких проблем не возникает:

```
SELECT DISTINCT p.pub_name, a.au_name
FROM dbo.fnSalesCount(25) AS a
JOIN titleauthor AS ta
   ON a.au_id = ta.au_id
JOIN titles AS t
   ON ta.title_id = t.title_id
JOIN publishers AS p
   ON t.pub_id = p.pub_id
```

В результате выполнения этого оператора формируется список интересующих нас авторов с указанием всех тех издателей, которые опубликовали их книги (табл. 13.3).

Таблица 13.3. Результаты применения функции `fnSalesCount()` в сложном запросе с операторами соединения

Столбец `pub_name`	Столбец `au_name`
Algodata Infosystems	Carson, Cheryl
Binnet & Hardley	DeFrance, Michel
Algodata Infosystems	Dull, Ann
Algodata Infosystems	Green, Marjorie
New Moon Books	Green, Marjorie
Algodata Infosystems	Hunter, Sheryl
Algodata Infosystems	MacFeather, Stearns
Binnet & Hardley	MacFeather, Stearns
Algodata Infosystems	O'Leary, Michael
Binnet & Hardley	O'Leary, Michael
Binnet & Hardley	Panteley, Sylvia
New Moon Books	Ringer, Albert
Binnet & Hardley	Ringer, Anne
New Moon Books	Ringer, Anne

Вполне очевидно, что в приведенном выше операторе функция участвовала в операции соединения таким образом, как если бы она была таблицей или представлением. Единственное видимое различие состоит в том, что после имени функции указаны круглые скобки и задан параметр.

Даже если бы возможности пользовательских функций не выходили за рамки описанных выше, и это было бы просто замечательно, но иногда для решения поставленной задачи невозможно ограничиться лишь единственным оператором `SELECT`. В некоторых случаях требуются такие функции, которые уже не подлежат замене с помощью параметризованного представления. И действительно, даже на примере некоторых описанных выше скалярных функций в определенных обстоятельствах для получения необходимых результатов может потребоваться выполнить несколько операторов. Пользовательские функции вполне обеспечивают реализацию такого подхода. Безусловно, как показывает приведенный выше пример функции с одним оператором, ничто не препятствует использованию функций для формирования и возврата таблиц, созданных с помощью нескольких операторов. Единственное значительное различие между функциями с одним и несколькими операторами состоит в том, что в последнем случае необходимо присвоить возвращаемой таблице имя и

определить ее метаданные (во многом аналогично тому, как при использовании временных таблиц).

В данном примере приходится сталкиваться с одной из очень распространенных проблем в мире реляционных баз данных — с задачей построения иерархий. Предположим, что отдел кадров компании Northwind обращается к вам с просьбой решить следующую задачу. В базе данных Northwind компании Northwind имеется таблица Employees, на которой задана односторонняя связь, определенная на столбце ReportsTo для каждого служащего и показывающая, кому подчиняется данный служащий. Это означает, что установить связь между подчиненным и его непосредственным руководителем можно, связав идентификатор служащего, хранящийся в столбце ReportsTo, с идентификатором другого служащего, который хранится в столбце EmployeeID. В отделах кадров очень часто возникает необходимость сформировать на основании данных о непосредственной подчиненности служащих иерархическое дерево подчиненности, т.е. представленные в виде организационной схемы списки всех сотрудников, которые подчиняются прямо или косвенно тому или иному руководителю.

На первый взгляд такая задача кажется довольно простой, ведь если требуется составить список всех служащих, которые подчиняются, скажем, служащему Andrew Fuller, то можно написать примерно такой запрос, в котором таблица Employees соединяется сама с собой:

```
Use Northwind
SELECT Emp.EmployeeID, Emp.LastName, Emp.FirstName, Emp.ReportsTo
FROM Employees AS Emp
JOIN Employees AS Mgr
   ON Mgr.EmployeeID = Emp.ReportsTo
WHERE Mgr.LastName = 'Fuller'
   AND Mgr.FirstName = 'Andrew'
```

Опять-таки, на первый взгляд, может показаться, что по условиям задачи должны быть получены примерно такие результаты:

```
EmployeeID       LastName         FirstName        ReportsTo
-------------    -------------    -------------    -------------
1                Davolio          Nancy            2
3                Leverling        Janet            2
4                Peacock          Margaret         2
5                Buchanan         Steven           2
8                Callahan         Laura            2
(5 row(s) affected)
```

Но в действительности на этом задача не исчерпывается. Дело в том, что должна быть получена информация обо всех служащих, которые принадлежат к дереву подчиненности, начинающемуся от служащего Andrew Fuller и включающему не только тех, кто подчиняется служащему Andrew Fuller, но и тех, кто подчиняется служащим, подчиняющимся служащему Andrew Fuller, и т.д. В частности, просмотр всех строк в таблице Employees базы данных Northwind показывает, что многие служащие подчиняются служащему Steven Buchanan, но информация о них отсутствует в результатах данного запроса.

Как показывает преподавательский опыт автора, наиболее сообразительные или опытные студенты тут же заявляют, что в этом нет никаких проблем, поскольку достаточно просто еще раз включить таблицу `Employees` *в операцию соединения. Безусловно, подобное решение было бы осуществимо при наличии весьма небольшого набора данных или в любой другой ситуации, когда количество уровней иерархии ограничено, но такие условия складываются далеко не всегда. Вполне может быть так, что в компании есть служащие, подчиняющиеся служащему* `Steven Buchanan`*, а также другие служащие, подчиняющиеся служащим, которые подчиняются служащему* `Steven Buchanan`*, и т.д., иными словами, глубина дерева подчиненности может увеличиваться практически неограниченно. Поэтому необходимо найти другой подход.*

В действительности требуется функция, которая возвращала бы информацию обо всех уровнях иерархии, расположенных ниже заданного значения идентификатора служащего `EmployeeID` (следовательно, идентификатора служащего, выполняющего роль руководителя, — `ManagerID`). Способ, позволяющий наилучшим образом решить эту задачу, представляет собой классический пример **рекурсии**. Если в каком-то блоке кода вызывается сам этот код, такой вызов рассматривается как рекурсивный. В предыдущей главе уже рассматривались такие примеры рекурсивного кода, как хранимые процедуры `spFactorial` и `spTriangular`. А в данном случае возникает задача, алгоритм решения которой может выглядеть так, как описано ниже.

1. Составить список всех служащих, которые подчиняются интересующему нас служащему, выполняющему роль руководителя.
2. Для каждого служащего, внесенного в список в шаге 1, составить список подчиняющихся ему служащих.
3. Повторять шаг 2 до тех пор, пока не удастся больше найти служащих, подчиняющихся кому-либо из служащих, внесенных в списки в ходе выполнения предыдущих шагов.

Приведенная формулировка представляет собой классический пример рекурсии. Это означает, что для обеспечения работы функции необходимо использовать операторы нескольких типов: одни из них должны обеспечивать определение того, какой уровень должен стать текущим, а другие (по меньшей мере один) должны снова вызывать ту же функцию для перехода на очередной, более низкий уровень иерархии.

Следует учитывать, что на пользовательские функции распространяются те же ограничения на пределы рекурсии, что и на хранимые процедуры. Это означает, что допускается переход не больше чем на 32 уровня рекурсии, поэтому если возникает вероятность достижения указанного предела, то при создании кода приходится применять определенный творческий подход для предотвращения ошибок.

Реализуем описанный замысел рекурсивного алгоритма в виде функции. Следует отметить, что в объявлении этой функции внесено несколько изменений. Дело в том, что на этот раз требуется связать с возвращаемым значением имя переменной (в данном случае `@Reports`), поскольку к нему приходится обращаться каждый раз, когда для выработки результата могут использоваться различные операторы. Кроме того, необходимо объявить возвращаемую таблицу; это позволяет СУБД SQL Server определить, предпринимается ли попытка вставки данных в эту таблицу перед ее возвратом в вызывающую процедуру:

```
CREATE FUNCTION dbo.fnGetReports
      (@EmployeeID AS int)
      RETURNS @Reports TABLE
      (
      EmployeeID     int         NOT NULL,
      ReportsToID    int         NULL
      )
AS
BEGIN
/* Эта функция должна вызываться рекурсивно, по одному разу для каждого служащего,
** играющего роль подчиненного (чтобы убедиться в том, что он не рассматривается
** как подчиненный по отношению к самому себе), поэтому требуется переменная,
** позволяющая следить за тем, данные о каком служащем рассматриваются
** в данный момент */
DECLARE @Employee AS int
/* В следующем операторе производится вставка данных о рассматриваемом
** служащем в рабочую таблицу. Важно отметить, что первая запись должна служить
** чем-то вроде образца для рекурсивной функции, и этот образец создается */
INSERT INTO @Reports
   SELECT EmployeeID, ReportsTo
   FROM Employees
   WHERE EmployeeID = @EmployeeID
/* Теперь производится выборка образца для рекурсивных вызовов. Для этого,
** по-видимому, было бы лучше применить курсор, но речь об этом пойдет в одной
** из следующих глав... */
SELECT @Employee = MIN(EmployeeID)
FROM Employees
WHERE ReportsTo = @EmployeeID
/* Следующие операции также, по-видимому, было бы лучше выполнить с помощью
** курсора, но на данный момент эта тема еще не пройдена, поэтому соответствующие
** действия просто моделируются. Обратите внимание на то, что вызов функции
** является рекурсивным! */
WHILE @Employee IS NOT NULL
   BEGIN
      INSERT INTO @Reports
          SELECT *
          FROM fnGetReports(@Employee)
          SELECT @Employee = MIN(EmployeeID)
          FROM Employees
          WHERE EmployeeID > @Employee
             AND ReportsTo = @EmployeeID
   END
RETURN
END
GO
```

В определении функции, приведенном выше, предусмотрено получение лишь минимальной информации о служащем и его руководителе, поскольку, если бы потребовалось получить дополнительную информацию, можно было бы просто снова выполнить соединение с таблицей `Employees`. Кроме того, автор позволил себе немного расширить рамки толкования требований, предъявленных в условиях задачи, поэтому включил в результаты и данные о самом указанном руководителе. Это было сделано в основном для упрощения реализации рекурсивного алгоритма, а также для предоставления своего рода исходного результата для результирующего набора.

В качестве иллюстрации рассмотрим пример формируемых результатов — служащий Andrew Fuller имеет идентификатор служащего EmployeeID, равный 2, поэтому непосредственно укажем данный идентификатор в вызове функции:

```
SELECT * FROM fnGetReports(2)
```

В результате выполнения этого запроса будет получена не только ранее выявленная информация о пяти служащих, подчиняющихся служащему Andrew Fuller, но также информация о тех, кто подчиняется служащему Steven Buchanan (который подчиняется мистеру Фуллеру) и о самом мистере Фуллере (напомним, что было решено включать в качестве исходной точки информацию о служащем, находящемся на самой вершине дерева подчиненности):

```
EmployeeID             ReportsToID
---------------        -----------------
2                      NULL
1                      2
3                      2
4                      2
5                      2
6                      5
7                      5
9                      5
8                      2
(9 row(s) affected)
```

Как оказалось, в полученные результаты вошли сведения обо всех служащих компании Northwind (эти результаты на компьютере читателя могут оказаться другими, если была также введена информация о других служащих), причем проверка вручную полученных данных с помощью информации, представленной в столбце ReportsTo, показывает, что полученные результаты действительно соответствуют ожидаемым. Но чтобы выполнить еще одну проверку, можно вызвать рассматриваемую функцию, указав идентификатор служащего Steven Buchanan (который равен 5):

```
SELECT * FROM fnGetReports(5)
EmployeeID             ReportsToID
---------------        ----------------
5                      2
6                      5
7                      5
9                      5
(4 row(s) affected)
```

Как и следовало ожидать, объем полученных результатов уменьшился. Теперь мы можем перейти к осуществлению завершающего шага и применить операцию соединения к полученным и исходным данным. Для этого воспользуемся почти таким же запросом, с помощью которого была предпринята первая попытка выяснить, кто является подчиненным служащего Andrew Fuller:

```
DECLARE @EmployeeID int
SELECT @EmployeeID = EmployeeID
    FROM Employees
    WHERE LastName = 'Fuller'
    AND FirstName = 'Andrew'
```

```
SELECT Emp.EmployeeID, Emp.LastName, Emp.FirstName, Mgr.LastName AS ReportsTo
FROM Employees AS Emp
JOIN dbo.fnGetReports(@EmployeeID) AS gr
  ON gr.EmployeeID = Emp.EmployeeID
JOIN Employees AS Mgr
  ON Mgr.EmployeeID = gr.ReportsToID
```

В результате будет получена информация обо всех восьми служащих, которые прямо или косвенно подчиняются мистеру Фуллеру:

```
EmployeeID       LastName         FirstName        ReportsTo
-------------    -------------    -------------    --------------
1                Davolio          Nancy            Fuller
3                Leverling        Janet            Fuller
4                Peacock          Margaret         Fuller
5                Buchanan         Steven           Fuller
6                Suyama           Michael          Buchanan
7                King             Robert           Buchanan
9                Dodsworth        Anne             Buchanan
8                Callahan         Laura            Fuller
(8 row(s) affected)
```

Любопытно отметить, что в результатах выполнения последнего запроса отсутствует информация о самом мистере Фуллере, хотя, как уже было сказано, информация о нем присутствует в результатах выполнения функции. Причина отсутствия данных о служащем `Andrew Fuller` состоит в том, что в результирующем наборе значение поля `ReportsTo` в строке с данными об этом служащем составляет `NULL`, поэтому отсутствует информация, на основании которой могло бы быть выполнено соединение с таблицей `Employees`. Таким образом, исключение данных об этом служащем происходит на этапе выполнения запроса, а не функции.

Таким образом, приведенный выше пример показывает, что пользовательские функции позволяют использовать для формирования табличных результатов очень сложный код, но поскольку полученные результаты представлены в виде таблицы, то их можно использовать для осуществления дальнейших операций наравне с любой другой таблицей.

Требования по обеспечению детерминированного выполнения функций

Одним из важных требований к пользовательским функциям является обеспечение их детерминированного выполнения. До сих пор в данной книге требования по обеспечению детерминированного выполнения рассматривались применительно к тому, как с помощью СУБД SQL Server осуществляется поиск данных в таблице, на которой задан индекс. В соответствии с этими требованиями индекс должен определять каждый индексируемый им элемент данных детерминированно (т.е. полностью однозначно). А что касается функций, то требования по обеспечению их детерминированного выполнения предъявляются в связи с тем, что некоторые функции предоставляют данные для выполнения операций с индексированными объектами (например, с вычисленными столбцами или индексированными представлениями).

По указанному признаку пользовательские функции подразделяются на две категории – детерминированные и недетерминированные. Детерминированное поведение функции зависит скорее не от того, каковы ее параметры, а от действий, выполняемых в самой функции. Если функция возвращает одно и то же значение после каждого вызова с одним и тем же набором допустимых параметров, эта функция называется *детерминированной*. Примером детерминированной встроенной функции может служить `SUM()`. Сумма чисел 3, 5 и 10, полученная с помощью этой функции, всегда равна 18. С другой стороны, функция `GETDATE()` является *недетерминированной*, поскольку возвращаемые ею результаты изменяются после каждого вызова.

Функция рассматривается как детерминированная, если она соответствует четырем описанным ниже критериям.

- Функция должна быть привязанной к схеме. Это означает, что все объекты, от которых зависит функция, должны иметь зарегистрированную зависимость и не допускается внесение изменений в определения этих объектов без предварительного уничтожения зависимой от них функции.
- Все другие функции, которые ссылаются на рассматриваемую функцию, также должны быть детерминированными, независимо от того, являются ли они определяемыми пользователем или определены в системе.
- В функции нельзя ссылаться на таблицы, которые определены вне самой функции (но использование переменных типа таблицы и временных таблиц допускается при условии, что эти переменные типа таблицы или временные таблицы объявлены в области определения функции).
- В функции нельзя применять расширенную хранимую процедуру.

В том, насколько важным является требование по обеспечению детерминированного выполнения, можно сразу же убедиться при попытке сформировать индекс на представлении или вычисленном столбце. Создание индексов на представлениях или вычисленных столбцах допускается, только если есть возможность надежно определить результат выборки данных из представления или вычисленного столбца. Это означает, что при наличии в представлении или вычисленном столбце ссылки на недетерминированную функцию не будет разрешено создание индекса на этом представлении или столбце. Безусловно, возникающая при этом ситуация не является безвыходной, но она вынуждает разработчика предпринимать дополнительные действия для определения того, является ли функция детерминированной или не недетерминированной, прежде чем приступать к созданию индексов на представлениях или столбцах, в которых используется эта функция.

Это означает, что разработчику часто приходится решать важную задачу по определению того, является ли разрабатываемая им функция детерминированной или недетерминированной. При этом следует учитывать, что, кроме проверки соответствия создаваемой функции описанным выше критериям, можно также прибегнуть к помощи СУБД SQL Server, которая сообщает, является ли функция детерминированной или недетерминированной, поскольку информация об этом сохраняется в свойстве `IsDeterministic` интересующего вас объекта. Для проверки этой информации можно воспользоваться функцией `OBJECTPROPERTY`. Например, можно проверить детерминированность функции `DayOnly`, которая использовалась выше в данной главе, следующим образом:

```
USE Accounting
SELECT OBJECTPROPERTY(OBJECT_ID('DayOnly'), 'IsDeterministic')
```

Для многих программистов окажется неожиданным (а для многих, возможно, нет), что в ответе, полученном от СУБД, будет указано, что данная функция является недетерминированной:

```
-----------
0
(1 row(s) affected)
```

Проверьте указанный список критериев соответствия требованиям к детерминированной функции, чтобы узнать, сможете ли вы сами определить, почему данная функция не рассматривается как детерминированная.

Работая над этим примером, автор получил одно из тех не очень приятных напоминаний о том, насколько трудно обойтись без элементарных ошибок. Безусловно, я был уверен, что эта функция должна быть детерминированной, но она отнюдь не стала таковой по определению. Просидев за работой без сна слишком много ночей и уходя на отдых только в предрассветные часы, я полностью забыл выполнить очевидное требование – ввести опцию `WITH SCHEMABINDING`.

К счастью, существует возможность легко исправить тот единственный недостаток, который обнаруживается в определении функции `DayOnly`. Достаточно лишь ввести в определение функции опцию `WITH SCHEMABINDING`, и полученные результаты станут совсем другими:

```
ALTER FUNCTION DayOnly(@Date datetime)
RETURNS varchar(12)
WITH SCHEMABINDING
AS
BEGIN
   RETURN CONVERT(varchar(12), @Date, 101)
END
```

Теперь вызовем повторно на выполнение тот же запрос с функцией `OBJECTPROPERTY`, после чего будут получены данные, свидетельствующие о том, что теперь эта функция является детерминированной:

```
-----------
1
(1 row(s) affected)
```

Однако попытка применить такую же проверку к описанной выше функции `AveragePrice`, которая была создана в базе данных `pubs`, приводит к получению совсем других результатов. Эта функция выглядела примерно таким образом:

```
USE Pubs
GO
CREATE FUNCTION dbo.AveragePrice()
RETURNS money
WITH SCHEMABINDING
AS
BEGIN
   RETURN (SELECT AVG(Price) FROM dbo.Titles)
END
```

В функции `AveragePrice` привязка к схеме была осуществлена с самого начала, поэтому рассмотрим, какие результаты дает применение функции `OBJECTPROPERTY`:

```
SELECT OBJECTPROPERTY(OBJECT_ID('AveragePrice'), 'IsDeterministic')
```

Оказывается, что результаты выполнения функции `OBJECTPROPERTY` свидетельствуют о том, что функция `AveragePrice` является недетерминированной, даже несмотря на то, что она связана со схемой. Дело в том, что в данной функции применяется ссылка на таблицу, не являющуюся локальной по отношению к самой функции (на временную таблицу или на переменную типа таблицы, созданную вне функции).

Следует также отметить, что недетерминированной является и функция `PriceDifference`, описанная в том же разделе, что и функция `AveragePrice`. Безусловно, одна из причин этого заключается в том, что в определении функции `PriceDifference` не предусмотрено ее связывание со схемой, но еще важнее то, что в функции `PriceDifference` применяется ссылка на функцию `AveragePrice`. Как уже было сказано, если создаваемая функция ссылается на недетерминированную функцию, то по определению сама становится недетерминированной.

Отладка пользовательских функций

По существу, процесс отладки пользовательских функций весьма напоминает процесс отладки хранимых процедур, пример которого приведен в главе 12.

Откройте в программе Visual Studio окно Server Explorer, чтобы установить соединение и перейти к пользовательской функции. Щелкните правой кнопкой мыши на обозначении этой функции и выберите во всплывающем меню команду `Step Into`; выполняемые в ходе отладки действия являются полностью одинаковыми, за исключением того, что сам отлаживаемый программный объект берется из другого списка (из списка функций, а не хранимых процедур).

Применение инфраструктуры .NET для работы с базами данных

Как было описано в главе 12, в версии SQL Server 2005 предусмотрена возможность использовать сборки .NET в хранимых процедурах и функциях. Благодаря этому чрезвычайно расширились возможности не только хранимых процедур, но и функций.

Автор исходит из того, что большинство читателей настоящей книги относятся к категории начинающих разработчиков, поэтому полностью представляют себе, насколько трудно объяснить все последствия, связанные с применением возможностей инфраструктуры .NET для доступа к базам данных. В действительности прибегать к использованию всех открывающихся при этом возможностей приходится не слишком часто, но когда возникает такая необходимость, в большинстве случаев удается достичь весьма впечатляющих результатов. В частности, средства .NET позволяют реализовывать сложные формулы вычисления специальных алгоритмов, дают возможность обращаться к внешним источникам данных (например, к базам данных компаний, которые берут на себя обязанности по авторизации кредитных карточек и выполняют тому подобные функции), обеспечивают доступ к другим структурированным источникам данных, и т.д. Короче говоря, благодаря применению инфраструктуры .NET внезапно становятся относительно осуществимыми такие операции обработки данных, которые до сих пор было либо невозможно реализовать с помощью

программного обеспечения для баз данных, либо для этого приходилось выполнять чрезвычайно трудоемкую разработку.

Подтверждением сказанного может отчасти служить приведенный в данной главе пример реализации сложной формулы с помощью пользовательской функции. Еще одним направлением использования открывающихся возможностей может стать обработка табличных данных из внешних источников, скажем, представленных в формате CSV (Comma-Separated Value – значения, разделенные запятыми) или в каком-то подобном формате, с помощью сборки .NET, оформленной в виде функции СУБД SQL Server.

Однако вся тематика применения сборок .NET в СУБД SQL Server остается весьма сложной, поэтому дальнейшее ее описание будет продолжено в книге по SQL Server 2005 для профессионалов. Несмотря на сказанное, отметим, что в данной главе приведены важные и достаточно полные сведения, позволяющие успешно подготовиться к освоению этих новых и перспективных возможностей.

Резюме

Очевидно, что для описания пользовательских функций, приведенного в этой главе, не потребовалось вводить большой объем нового учебного материала. Фактически при создании пользовательских функций используются в основном такие же операторы и переменные, а также реализуются такие же процедуры кодирования, как и во время разработки сценариев и хранимых процедур. Тем не менее пользовательские функции открывают возможность реализации новых и очень привлекательных функциональных средств, которое не могли быть ранее осуществлены в СУБД SQL Server. Особенно важно то, что пользовательские функции позволяют намного расширить область применения программного обеспечения и даже допускают возможность их вложения непосредственно в запросы. К тому же с помощью пользовательских функций могут также формироваться параметризованные представления и динамически создаваемые таблицы.

В целом можно сделать вывод, что пользовательские функции относятся к числу наиболее перспективных программных средств, которые были введены в последних версиях SQL Server. Очевидно, что в одной короткой главе невозможно было раскрыть весь потенциал пользовательских функций, поэтому мне остается лишь выразить надежду на то, что читатель сумеет внести свой вклад в разработку новых способов их применения.

Упражнения

13.1. Повторно реализуйте сценарий `spTriangular`, рассматриваемый в главе 12, но на этот раз в виде функции, а не хранимой процедуры.