# ГЛАВА 13

# Пользовательские функции

Пользовательские функции принадлежат к числу наиболее привлекательных объектов SQL Server. Возможность применения пользовательских функций (User Defined Function — UDF) появилась уже достаточно давно, 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 в версии SQL Server 2005 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 не только приведено вводное описание пользовательских функций, но и 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 собой 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).

Причем даже при использовании выходного параметра, возвращающего табличное значение, остается необходимость сделать по крайней мере еще один дополнительный шаг, чтобы воспользоваться полученными результатами в запросе.

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 функций, 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 для разработчиков имеет особое значение то, что пользовательские функции обладают еще одним важным свойством. Речь идет о том, что пользовательские функции обеспечивают возврат не только скалярных значений, но и таблиц. 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>.]<data type> [ = <default value>
[READONLY]]
    [ ,...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 программировании. Во многом аналогично собственным встроенным функциям SQL Server (таким 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 главе 5. Предположим, что необходимо собрать сведения обо всех заказах, полученных за сегодняшний день. Начнем с того, что внесем в список несколько заказов, в которых проставлена сегодняшняя дата. Для этого 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 = 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, в течение одной миллисекунды, если используется полный формат datetime, а также в пределах 100 миллисекунд применительно к формату datetime2).

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)
```

*Безусловно, следует отметить, что такой же результат можно было бы получить путем приведения значения `@Date` к типу данных `date`. Но автор решил воспользоваться в данном случае функцией `CONVERT`, исключительно ради демонстрации способа выделения даты из данных о дате и времени, совместимого с предыдущими версиями (в SQL Server 2005 и более ранних версиях тип данных `date` не поддерживался).*

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. 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 date)
RETURNS date
AS
BEGIN
  RETURN @Date;
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.

Заслуживает внимания то, что предыдущая версия совместима с SQL Server 2008, поскольку в ней для исключения данных о времени предусмотрено приведение параметра к типу данных `date`. Если бы потребовалось обеспечить усечение полученных значений даты и времени с применением такого способа, который поддерживается в версии SQL Server 2005 (как в приведенном выше примере на основе запроса), то нужно было бы, как и раньше, прибегнуть к использованию функции `CONVERT`. Такой пример приведен в следующем операторе:

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

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. Причем очевидно, что даже в случае простого запроса, подобного этому, создается код нового типа, гораздо более удобный для чтения. А применяемый при этом вызов действует в основном по такому же принципу, как 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 функций со скалярными значениями, 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 AdventureWorks2008;

SELECT Name,
     ListPrice,
     (SELECT AVG(ListPrice) FROM Production.Product) AS Average,
      ListPrice - (SELECT AVG(ListPrice) FROM Production.Product)
       AS Difference
FROM Production.Product
WHERE ProductSubCategoryID = 1; -- Подкатегория Mountain Bikes
```

Выполнение приведенного кода приводит к получению довольно простого набора данных:

```
Name                        ListPrice    Average         Difference
--------------------------  ----------   -----------     --------------
Mountain-100 Silver, 38     3399.99      438.6662        2961.3238
Mountain-100 Silver, 42     3399.99      438.6662        2961.3238
Mountain-100 Silver, 44     3399.99      438.6662        2961.3238
Mountain-100 Silver, 48     3399.99      438.6662        2961.3238
Mountain-100 Black, 38      3374.99      438.6662        2936.3238
Mountain-100 Black, 42      3374.99      438.6662        2936.3238
:
:
Mountain-500 Silver, 52     564.99       438.6662        126.3238
Mountain-500 Black, 40      539.99       438.6662        101.3238
Mountain-500 Black, 42      539.99       438.6662        101.3238
Mountain-500 Black, 44      539.99       438.6662        101.3238
Mountain-500 Black, 48      539.99       438.6662        101.3238
Mountain-500 Black, 52      539.99       438.6662        101.3238

(32 row(s) affected)
```

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 представляет операцию вычисления среднего, а вторая — операцию вычитания:

```
CREATE FUNCTION dbo.AveragePrice()
RETURNS money
WITH SCHEMABINDING
AS
BEGIN
  RETURN (SELECT AVG(ListPrice) FROM Production.Product);
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.

*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 AdventureWorks2008

SELECT Name,
     ListPrice,
     dbo.AveragePrice() AS Average,
     dbo.PriceDifference(ListPrice) AS Difference
FROM Production.Product
WHERE ProductSubCategoryID = 1; -- Подкатегория Mountain Bikes
```

В результате формируются те же результаты, что и при использовании подзапроса.

Следует отметить, что пользовательские функции не только способствуют повышению удобства чтения кода, но и предоставляют дополнительное преимущество, связанное с многократным 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, являются не очень 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 AdventureWorks2008
GO

CREATE FUNCTION dbo.fnContactList()
RETURNS TABLE
AS
RETURN (SELECT BusinessEntityID,
        LastName + ', ' + FirstName AS Name
     FROM Person.Person);
GO
```

В этой функции выполняется возврат таблицы, состоящей из строк, полученных с помощью оператора `SELECT`, а также несложное форматирование — конкатенация фамилии и имени с разделением их запятыми.

С этого момента созданная функция может использоваться по такому же принципу, как таблица:

```
SELECT *
FROM dbo.fnContactList();
```

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 иногда возникает необходимость использовать в операторе выборки данных с помощью представления конкретные параметры. Например, может потребоваться передавать в запрос данные о фамилии, чтобы обеспечить выборку информации по условию (и избавиться тем самым от необходимости каждый раз вставлять вручную соответствующую конструкцию `WHERE`). Для решения этой задачи может применяться такой код:

```
-- Создание представления с помощью оператора CREATE
CREATE VIEW vFullContactName
AS
SELECT p.BusinessEntityID,
```

```
        LastName + ', ' + FirstName AS Name,
        ea.EmailAddress
     FROM Person.Person as p
     LEFT OUTER JOIN Person.EmailAddress ea
      ON ea.BusinessEntityID = p.BusinessEntityID;
GO
```

Разумеется, это позволяет решить поставленную задачу, но с учетом определенного нюанса. Дело в том, что невозможно предусмотреть применение параметров непосредственно в самом представлении, поэтому в запрос придется ввести конструкцию `WHERE`:

```
SELECT *
FROM vFullContactName
WHERE Name LIKE 'Ad%';
```

Выполнение этого оператора приводит к получению таких результатов:

```
BusinessEntityID Name                      EmailAddress
-------------------------------------------------------------------------------
67          Adams, Jay                  jay0@adventure-works.com
301          Adams, Frances               frances0@adventure-works.com
305          Adams, Carla                carla0@adventure-works.com
:
:
16901        Adams, Adam                 adam46@adventure-works.com
16902        Adams, Eric                 eric57@adventure-works.com
16910        Adams, Jackson               jackson47@adventure-works.com

(87 row(s) affected)
```

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

```
USE AdventureWorks2008;
GO

CREATE FUNCTION dbo.fnContactSearch(@LastName nvarchar(50))
RETURNS TABLE
AS
RETURN (SELECT p.BusinessEntityID,
        LastName + ', ' + FirstName AS Name,
        ea.EmailAddress
     FROM Person.Person as p
     LEFT OUTER JOIN Person.EmailAddress ea
      ON ea.BusinessEntityID = p.BusinessEntityID
     WHERE LastName Like @LastName + '%');
GO
```

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

```
SELECT *
FROM fnContactSearch('Ad');
```

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.

Даже если бы возможности пользовательских функций не выходили за рамки описанных выше, и это было бы просто замечательно. Однако иногда для решения поставленной задачи требуется нечто большее по сравнению с выполнением единственного оператора SELECT, как в приведенном примере. В некоторых случаях требуются такие функции, которые вообще 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, 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).

Чтобы проиллюстрировать этот пример, обсудим одну из весьма распространенных проблем в мире реляционных баз данных — задачу обработки иерархических данных.

Предположим, что от отдела кадров некоторой компании поступила просьба решить следующую задачу. В базе данных этой компании имеется таблица Employees, на которой задана односторонняя связь (внешний ключ, который связан с другим столбцом в той же таблице), определенная на столбце ManagerID для каждого служащего и показывающая, кому подчиняется данный служащий. Это означает, что можно установить связь между подчиненным и его непосредственным руководителем, связав идентификатор служащего, хранящийся в столбце ManagerID, 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.

Важным недостатком реляционных баз данных с первого момента их возникновения было то, что они не предоставляли мощных средств для работы с иерархическими данными. Этой теме посвящены многочисленные статьи, технические документы и книги. Но, к счастью, в СУБД SQL Server 2008 представлена новая методология работы с иерархическими данными. Она опирается на такие вновь введенные средства, как тип данных hierarchyID и коллекция встроенных функций, позволяющие существенно упростить обработку древовидных структур данных в реляционной базе данных. Эти новые средства являются довольно сложными, и для овладения ими требуются значительные усилия, поэтому мною было решено отложить описание связанной с ними тематики. Я решил рассматривать средства работы с иерархическими структурами как выходящие за рамки настоящей книги и поместить их описание, наряду с описанием нового типа данных HierarchyID, в главу подготавливаемой мною книги для профессионалов, посвященную усовершенствованным структурам данных.

*С примерами новых функциональных средств, предназначенных для работы с иерархическими структурами данных, которые вошли с состав версии SQL Server 2008, можно ознакомиться, рассмотрев столбцы* `OrganizationNode` *и* `OrganizationLevel` *таблицы HumanResources.*`Employee` *в базе данных* `AdventureWorks2008`.

А пока, чтобы продолжить начатое в этой главе содержательное обсуждение иерархических данных, воспользуемся давно сложившимся методом представления и обработки иерархий, который принято называть "традиционным методом". В базе данных `AdventureWorks2008` отсутствуют примеры применения этого более старого (и все еще гораздо более распространенного) подхода к работе с иерархическими данными, поэтому создадим собственную версию таблицы `Employee` (и назовем ее `Employee2`), в которой будет реализован "традиционный метод" поддержки иерархий. Если читатель выполнил сценарий `BuildAndPopulateEmployee2.sql`, приведенный в главе 3, то уже имеет в своем распоряжении новую версию таблицы `Employee`. Если такая работа еще не проделана читателем, рекомендуем заняться этим и выполнить указанный сценарий на данном этапе (напомним, что требуемый для этого код можно найти на веб-сайтах, сопровождающих данную книгу, `wrox.com` и `professionalsql.com`).

Таблица, созданная с помощью этого сценария, представлена на рис. 13.1.

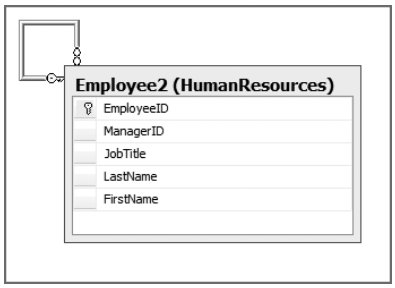


*Рис. 13.1. Структура таблицы, созданной с помощью сценария* `BuildAndPopulateEmployee2.sql`

После выполнения указанного сценария и создания таблицы `Employee2` можно приступить к решению, например, такой задачи: получение списка служащих, которые подчиняются Карле Хантингтон (`Karla Huntington`).

На первый взгляд эта задача кажется довольно несложной. Можно предположить, что для выявления всех служащих, для которых Карла является руководителем, можно написать запрос, соединяющий таблицу `Employee` с самой собой:

```
USE AdventureWorks2008;

SELECT
     TheReport.EmployeeID,
     TheReport.JobTitle,
     TheReport.LastName,
     TheReport.FirstName
FROM
     HumanResources.Employee2 as TheBoss
JOIN HumanResources.Employee2 AS TheReport
     ON TheBoss.EmployeeID = TheReport.ManagerID
WHERE TheBoss.LastName = 'Huntington' AND TheBoss.FirstName = 'Karla';
```

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

```
EmployeeID    JobTitle                    LastName       FirstName
----------    -------------------------   ---------      --------------------
5             VP of Engineering           Olsen          Ken
6             VP of Professional Services Cross          Gary
7             VP of Security              Lebowski       Jeff
(3 row(s) affected)
```

Но в действительности на этом проблема не исчерпывается. Дело в том, что фактически требуется найти всех сотрудников во всех цепочках подчиненности Карле, — не только тех, кто непосредственно подчиняется Карле, но и тех, кто подчиняется тем, кто подчиняется Карле, и т.д. В качестве примера следует отметить, что после просмотра всех строк во вновь созданной нами таблице `Employee2` можно обнаружить целый ряд сотрудников, которые подчиняются, например, Кэну Олсену (`Ken Olsen`), подчиненному Карлы, но данные о них не появляются в результатах приведенного выше запроса.

*Безусловно, читатели могут предположить, что решение такой уточненной задачи не представляет какой-либо сложности, поскольку для формирования следующего уровня подчиненности достаточно еще раз включить таблицу `Employee2` в операцию соединения.*

*Несомненно, 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. Предположим, что есть такие сотрудники, которые подчиняются Роберту Чичову (`Robert Cheechov`), подчиненному Кэна, а им, в свою очередь, подчиняются другие сотрудники, причем такие цепочки подчиненности могут приобретать неопределенно большую длину. Как же поступить в таком случае? Поиску ответа на этот вопрос будет посвящена остальная часть данного раздела.*

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`), т.е. функция, формирующая дерево. 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 рассматривался такой пример рекурсивного кода, как хранимая процедура `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 объявление 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,
    ManagerID  int      NULL
    )
AS
BEGIN

/* Эту функцию необходимо вызывать рекурсивно, по одному разу для каждого
** подчиняющегося сотрудника (чтобы узнать, имеет ли тот или иной сотрудник
** собственных подчиненных), поэтому требуется промежуточная переменная,
** позволяющая следить за тем, данные какого сотрудника обрабатываются
** в настоящее время.*/
DECLARE @Employee AS int;

/* С помощью этого оператора выполняется вставка данных о текущем сотруднике
** в рабочую таблицу. При этом важно то, что в этом случае первая строка должна
** играть роль своего рода точки отсчета, поскольку применяемая нами функция
** является рекурсивной; именно этим обусловлен выбранный способ получения
** данных этой строки. */
INSERT INTO @Reports
  SELECT EmployeeID, ManagerID
  FROM HumanResources.Employee2
  WHERE EmployeeID = @EmployeeID;

/* Теперь необходимо также сформировать точку отсчета для рекурсивных вызовов,
** к которым мы приступим с помощью этой функции. По-видимому, эту задачу было бы
** лучше всего решить с помощью курсора, но эта тема еще не была рассмотрена
** к тому времени, как мы приступили к этой главе...*/
SELECT @Employee = MIN(EmployeeID)
FROM HumanResources.Employee2
WHERE ManagerID = @EmployeeID;

/* Следующую часть работы, вероятно, удобнее было бы выполнить с применением
** курсора, но нам еще только предстоит изучить главу с описанием курсоров,
** поэтому организуем работу по принципу имитации их действия. Обратите внимание
** на то, что осуществляется рекурсивный вызов применяемой функции!*/
```

```
WHILE @Employee IS NOT NULL
  BEGIN
    INSERT INTO @Reports
       SELECT *
       FROM fnGetReports(@Employee);

       SELECT @Employee = MIN(EmployeeID)
       FROM HumanResources.Employee2
       WHERE EmployeeID > @Employee
         AND ManagerID = @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 `Employee2`. Кроме того, я 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 `Karla Huntington` имеет идентификатор служащего `EmployeeID`, равный 4, поэтому непосредственно укажем данный идентификатор в вызове функции:

```
SELECT * FROM fnGetReports(4);
```

В результате выполнения этого запроса будет получена не только ранее выявленная информация об одном служащем, подчиняющемся служащему `Karla Huntington`, но также информация о тех, кто подчиняется служащему `Ken Olsen` (который подчиняется госпоже Хантингтон), и о самой госпоже Хантингтон (напомним, что было решено включать в качестве исходной точки информацию о служащем, находящемся на самой вершине дерева подчиненности).

```
EmployeeID     ManagerID
-----------    -----------
4              1
5              4
8              5
9              5
10             5
11             5
6              4
7              4

(8 row(s) affected)
```

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 `Karla Huntington`:

```
DECLARE @EmployeeID int;

SELECT @EmployeeID = EmployeeID
FROM HumanResources.Employee2 e
WHERE LastName = 'Huntington'
AND FirstName = 'Karla';

SELECT e.EmployeeID, e.LastName, e.FirstName, m.LastName AS ReportsTo
FROM HumanResources.Employee2 AS e
JOIN dbo.fnGetReports(@EmployeeID) AS r
 ON e.EmployeeID = r.EmployeeID
JOIN HumanResources.Employee2 AS m
 ON m.EmployeeID = r.ManagerID;
```

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

```
EmployeeID    LastName        FirstName       ReportsTo
-----------   ------------    ------------    ---------------
4             Huntington      Karla           Smith
5             Olsen           Ken             Huntington
8             Gutierrez       Ron             Olsen
9             Bray            Marky           Olsen
10            Cheechov        Robert          Olsen
11            Gale            Sue             Olsen
6             Cross           Gary            Huntington
7             Lebowski        Jeff            Huntington

(8 row(s) affected)
```

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 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, то она называется детер-

минированной. Примером детерминированной встроенной функции может служить SUM(). Сумма чисел 3, 5 и 10 всегда равна 18, а функция SUM() при любом ее вызове с указанными параметрами возвращает указанное значение. С другой стороны, значение функции GETDATE() является недетерминированным, поскольку никто не гарантирует получение одного и того же значения при каждом ее вызове.

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 удаления 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. (Использование переменных типа таблицы вполне приемлемо. Временные таблицы допускаются при условии, что они объявлены в области определения функции.)
- В функции нельзя применять расширенные хранимые процедуры.

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.

В связи с этим мы должны научиться находить ответ на вопрос о том, является ли рассматриваемая функция детерминированной или нет. При этом следует учитывать, что, кроме проверки соответствия создаваемой функции описанным выше критериям, можно прибегнуть к помощи СУБД SQL Server, которая сообщает, является ли функция детерминированной или недетерминированной, поскольку информация об этом сохраняется в свойстве IsDeterministic интересующего нас 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');
```

На первый взгляд может показаться неожиданным, что эта функция на является детерминированной:

```
-----------
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, я 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 данной функции. Достаточно лишь ввести в определение функции опцию `WITH SCHEMABINDING`, и полученные результаты станут совсем другими:

```
ALTER FUNCTION dbo.DayOnly(@Date date)
RETURNS date
WITH SCHEMABINDING
AS
BEGIN
  RETURN @Date;
END
```

Теперь вызовем повторно на выполнение тот же запрос с функцией `OBJECTPROPERTY`:

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

Итак, теперь эта функция является детерминированной.

Однако попытка применить такую же проверку к описанной выше функции `Average Price`, которая была создана в базе данных `AdventureWorks2008`, приводит к получению совсем других результатов. Эта функция выглядела таким образом:

```
CREATE FUNCTION dbo.AveragePrice()
RETURNS money
WITH SCHEMABINDING
AS
BEGIN
  RETURN (SELECT AVG(ListPrice) FROM Production.Product);
END
GO

CREATE FUNCTION dbo.PriceDifference(@Price money)
RETURNS money
AS
BEGIN
  RETURN @Price - dbo.AveragePrice();
END
```

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

```
USE AdventureWorks2008;

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 Price Difference, описанная в том же разделе, что и функция AveragePrice. Безусловно, одна из причин этого заключается в том, что в определении функции Price Difference 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.

Подготовьте отдельный сценарий, в котором вызывается рассматриваемая функция, и пошагово выполняйте сценарий (с помощью значка на панели инструментов или нажатия клавиши <F11>). После этого можно выполнить шаг с заходом, при котором управление передается непосредственно в рассматриваемую пользовательскую функцию.

# Применение инфраструктуры .NET для работы с базами данных

Как было описано в главе 12, начиная с 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 представляет 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 реализовать сложные формулы вычисления специальных алгоритмов, причем без особых затруднений; 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 (а в некоторых случаях даже выходить за рамки допустимых способов применения программных средств).

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 2008 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 сам читатель со временем сумеет внести свой вклад в разработку новых способов их применения.

# Упражнение

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