Базы данных

Тема 10

СУБД SQL Server 2012 – Хранимые процедуры, курсоры, пользовательские функции.

Хранимые процедуры

- хранимые процедуры в Microsoft SQL Server аналогичны процедурам в других языках программирования:
  - обрабатывают входные аргументы и возвращают вызывающей процедуре или пакету значения в виде выходных аргументов (в том числе через параметры типа cursor);
  - содержат программные инструкции, которые выполняют операции в базе данных, в том числе вызывающие другие процедуры;
  - возвращают значение состояния (код результата) вызывающей процедуре или пакету, таким образом передавая сведения об успешном или неуспешном завершении;
- по сравнению с программами Transact-SQL, которые хранятся локально на клиентских компьютерах, хранимые процедуры SQL Server имеют следующие преимущества:
  - регистрируются на сервере;
  - могут иметь атрибуты безопасности;
  - позволяют сделать защиту приложений более надежной;
  - поддерживают модульное программирование;
  - представляют собой именованный код, дающий возможность отсроченного связывания;
  - позволяют уменьшить сетевой трафик.

Типы хранимых процедур SQL Server 2012

- **пользовательские хранимые процедуры**:
  - Transact-SQL: сохраненная коллекция инструкций языка Transact-SQL (инструкций языка описания данных (DDL) и языка обработки данных (DML)), которая может принимать и возвращать параметры;
  - CLR: ссылка на метод общеязыковой среды выполнения (CRL) платформы Microsoft .NET Framework, который может принимать и возвращать пользователю параметры;
- **расширенные хранимые процедуры** (устарели и не будут поддерживаться в следующих выпусках): библиотеки DLL, которые могут динамически загружаться и выполняться экземпляром Microsoft SQL Server;
- **системные хранимые процедуры**: специальные процедуры, позволяющие осуществлять административные действия в SQL Server 2012:
  - физически системные хранимые процедуры хранятся в базе данных ресурсов и имеют префикс **sp_...**;
  - логически же они отображаются в любой системной или пользовательской базе данных в схеме **sys**);
  - SQL Server поддерживает расширенные системные хранимые процедуры (имеют префикс **xp_**), обеспечивающие интерфейс между SQL Server и внешними программами для выполнения различных действий по обслуживанию системы.

Создание и удаление хранимых процедур

```
CREATE PROCEDURE procedure_name [ ; number ]
   [ { @parameter data_type }
     [ VARYING ] [ = default ] [ OUTPUT ]
   ] [ ,...n ]
AS
   sql_statement [ ...n ]
```

```
USE AdventureWorks;
GO
IF OBJECT_ID ('HumanResources.usp_GetAllEmployees', 'P' ) IS NOT NULL
DROP PROCEDURE HumanResources.usp_GetAllEmployees;
GO

CREATE PROCEDURE HumanResources.usp_GetAllEmployees AS
    SELECT LastName, FirstName, JobTitle, Department
    FROM HumanResources.vEmployeeDepartment;
GO

EXECUTE HumanResources.usp_GetAllEmployees;
GO
-- Or
EXEC HumanResources.usp_GetAllEmployees;
GO
-- Or, if this procedure is the first statement within a batch:
HumanResources.usp_GetAllEmployees;
```

Хранимые процедуры с подстановочными параметрами

```
IF OBJECT_ID ( 'HumanResources.usp_GetEmployees2', 'P') IS NOT NULL
   DROP PROCEDURE HumanResources.usp_GetEmployees2;
GO

CREATE PROCEDURE HumanResources.usp_GetEmployees2
   @lastname varchar(40) = 'D%',
   @firstname varchar(20) = '%'
AS
   SELECT LastName, FirstName, JobTitle, Department
   FROM HumanResources.vEmployeeDepartment
   WHERE FirstName
   LIKE @firstname AND LastName LIKE @lastname;
GO

EXECUTE HumanResources.usp_GetEmployees2 'Wi%';
-- Or
EXECUTE HumanResources.usp_GetEmployees2 @firstname = '%';
-- Or
EXECUTE HumanResources.usp_GetEmployees2 '[CK]ars[OE]n';
-- Or
EXECUTE HumanResources.usp_GetEmployees2 'Hesse', 'Stefen'
```

Возвращение данных с использованием параметров OUTPUT и кода возврата

- если при выполнении хранимой процедуры указано OUTPUT для параметра, а параметр не указан при помощи OUTPUT в хранимой процедуре, выдается сообщение об ошибке;
- можно выполнять хранимую процедуру с параметрами OUTPUT и не указывать OUTPUT при выполнении хранимой процедуры: сообщение об ошибке не будет выдаваться, но нельзя будет использовать выходное значение в вызывающей программе.

```
CREATE PROCEDURE Sales.usp_GetEmployeeSalesYTD
    @SalesPerson nvarchar(50),
    @SalesYTD money OUTPUT
AS
    SELECT @SalesYTD = SalesYTD
    FROM Sales.SalesPerson AS sp
    JOIN HumanResources.vEmployee AS e ON e.EmployeeID =
    sp.SalesPersonID
    WHERE LastName = @SalesPerson;
RETURN
GO

DECLARE @SalesYTDBySalesPerson money;
EXECUTE @result = Sales.usp_GetEmployeeSalesYTD
         N'Blythe', @SalesYTD = @SalesYTDBySalesPerson OUTPUT;
PRINT 'YTD sales' + convert(varchar(10),@SalesYTDBySalesPerson);
GO
```

Курсоры

- операции в реляционной базе данных выполняются над множеством строк. Приложения, особенно интерактивные, не всегда эффективно работают с результирующим набором (набором строк, возвращаемым инструкцией) *Курсоры* являются расширением результирующих наборов, которые предоставляют механизм, позволяющий обрабатывать одну строку или небольшое их число за один раз;
- курсоры обеспечивают возможность:
  - позиционирования на отдельные строки результирующего набора;
  - получения одной или нескольких строк от текущей позиции в результирующем наборе;
  - изменения данных в строках в текущей позиции результирующего набора.

Типы курсоров

- статические (static)
  - результирующий набор фиксируется при открытии курсора (не отражает изменения, влияющие на вхождение в результирующий набор, изменения значений, а также изменения в результате удаления строк в базе данных);
  - в Microsoft SQL Server 2012 доступны только для чтения;
- основанные на потенциальном ключе (keyset)
  - набор записей фиксируется при открытии: членство и порядок строк являются фиксированными при открытии курсора;
  - курсоры управляются с помощью набора уникальных идентификаторов – ключей, которые создаются из набора столбцов, уникально идентифицирующего строки результирующего набора;
  - изменения в значениях столбцов, не входящих в ключевой набор (внесенные владельцем курсора или другими пользователями), становятся видны при прокрутке курсора;
  - вставка в базу данных, выполненная вне курсора, видна только после его повторного открытия;
  - при попытке запросить состояние строки, удаленной после открытия курсора, функция @@FETCH_STATUS возвращает состояние "строка отсутствует“;
  - обновление ключевого столбца действует аналогично удалению старого значения ключа с последующей вставкой нового значения (если обновление не было выполнено с помощью курсора, то новое ключевое значение не будет видимо);
- динамические
  - набор записей не фиксируется при открытии курсора (отражаются все изменения строк в результирующем наборе при прокрутке курсора: значения типа данных, порядок и членство строк в результирующем наборе могут меняться для каждой выборки).

Обработка курсоров

- курсоры Transact-SQL и API-курсоры имеют различный синтаксис, но для всех курсоров SQL Server используется одинаковый цикл обработки:
  1. связать курсор с результирующим набором инструкции Transact-SQL и задать его характеристики (например, возможность обновления строк);
  2. выполнить инструкцию Transact-SQL для заполнения курсора;
  3. получить в курсор необходимые строки:
     1. операция получения в курсор одной и более строк называется выборкой;
     2. выполнение серии выборок для получения строк в прямом или обратном направлении называется прокруткой;
  4. при необходимости выполнить операции изменения (обновления или удаления) строки в текущей позиции курсора;
  5. закрыть курсор.

- *однонаправленный курсор* не поддерживает прокрутку, а только последовательную выборку строк от начала курсора до его конца:
  – строки не извлекаются из базы данных, пока они не будут выбраны;
  – результаты всех инструкций INSERT, UPDATE и DELETE, которые влияют на строки результирующего набора и выполненные текущим пользователем или зафиксированы другими пользователями, отображаются в процессе выборки строк из курсора.

Объявление курсора (Transact-SQL) и использование переменной типа *cursor*

```
DECLARE cursor_name CURSOR
   [ LOCAL | GLOBAL ]
   [ FORWARD_ONLY | SCROLL ]
   [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
   [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
   [ TYPE_WARNING ]
   FOR select_statement
   [ FOR UPDATE [ OF column_name [ ,...n ] ] ]

DECLARE @cursor_variable_name CURSOR

SET @cursor_variable_name = CURSOR
   [ FORWARD_ONLY | SCROLL ]
   [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
   [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
   [ TYPE_WARNING ]
   FOR select_statement
   [ FOR UPDATE [ OF column_name [ ,...n ] ] ]
```

Объявление курсора: примеры

```
/* Use DECLARE @local_variable, DECLARE CURSOR and SET. */
DECLARE @MyVariable CURSOR
DECLARE MyCursor CURSOR FOR
SELECT LastName FROM AdventureWorks.Person.Contact
SET @MyVariable = MyCursor
GO

/* Use DECLARE @local_variable and SET */
DECLARE @MyVariable CURSOR
SET @MyVariable = CURSOR SCROLL KEYSET FOR
SELECT LastName FROM AdventureWorks.Person.Contact;

DEALLOCATE MyCursor;
```

Работа с курсором

- открытие курсора

```
OPEN { { [ GLOBAL ] cursor_name }
      | @cursor_variable_name }
```

- позиционирование курсора

```
FETCH
    [ [ NEXT | PRIOR | FIRST | LAST
             | ABSOLUTE { n | @nvar }
             | RELATIVE { n | @nvar } ]
    FROM ]
     { { [ GLOBAL ] cursor_name } |
              @cursor_variable_name }
     [ INTO @variable_name [ ,...n ] ]
```

- закрытие курсора

```
CLOSE { { [ GLOBAL ] cursor_name }
           | @cursor_variable_name }
```

- удаление курсора

```
DEALLOCATE { { [ GLOBAL ] cursor_name }
       | @cursor_variable_name }
```

Пример использования курсора

```
DECLARE @name varchar(40)

DECLARE authors_cursor CURSOR
    FOR SELECT au_lname FROM authors

OPEN authors_cursor

FETCH NEXT FROM authors_cursor INTO @name

WHILE @@FETCH_STATUS = 0
BEGIN
    FETCH NEXT FROM authors_cursor INTO @name
END

CLOSE authors_cursor

DEALLOCATE authors_cursor
```

Изменение данных с использованием курсора

```
DECLARE complex_cursor CURSOR FOR
    SELECT a.BusinessEntityID
    FROM HumanResources.EmployeePayHistory AS a
    WHERE RateChangeDate <>
        (SELECT MAX(RateChangeDate) FROM
        HumanResources.EmployeePayHistory AS b
        WHERE a.BusinessEntityID = b.BusinessEntityID);

OPEN complex_cursor;
FETCH FROM complex_cursor;

DELETE FROM HumanResources.EmployeePayHistory
WHERE CURRENT OF complex_cursor;

CLOSE complex_cursor;
DEALLOCATE complex_cursor;
```

Использование типа данных cursor в параметре OUTPUT

- хранимые процедуры языка Transact-SQL могут использовать тип данных **cursor** только для параметров OUTPUT:
  - если тип данных **cursor** указан для параметра, должны быть также указаны оба параметра VARYING и OUTPUT;
  - если для параметра указано ключевое слово VARYING, тип данных должен быть **cursor** и должно быть указано ключевое слово OUTPUT;
- следующие правила относятся к выходным параметрам типа **cursor** при выполнении процедуры:
  - для однонаправленного курсора в результирующий набор курсора будут возвращены только строки с текущей позиции курсора до конца курсора - текущая позиция курсора определяется при окончании выполнения хранимой процедуры;
  - для прокручиваемого курсора, все строки в результирующем наборе будут возвращены к вызывающему пакету, хранимой процедуре или триггеру после выполнения хранимой процедуры - при возврате позиция курсора остается в позиции последней выборки, выполненной в процедуре;
  - для любого типа курсора, если курсор закрыт, то вызывающему пакету, хранимой процедуре или триггеру будет возвращено значение NULL, то же произойдет в случае, если курсор присвоен параметру, но этот курсор никогда не открывался.

Пример использования курсора

```
CREATE PROCEDURE dbo.currency_cursor
   @currency_cursor CURSOR VARYING OUTPUT
AS
   SET @currency_cursor = CURSOR
        FORWARD_ONLY STATIC FOR
        SELECT CurrencyCode, Name
        FROM Sales.Currency;
OPEN @currency_cursor;
GO

DECLARE @MyCursor CURSOR;
EXEC dbo.currency_cursor @currency_cursor = @MyCursor
   OUTPUT;
WHILE (@@FETCH_STATUS = 0)
BEGIN;
   FETCH NEXT FROM @MyCursor;
END;
CLOSE @MyCursor;
DEALLOCATE @MyCursor;
GO
```

Определяемые пользователем функции

- преимущества использования:
  - возможность модульного программирования;
  - ускорение выполнения за счет кэширования и повторного использования планов выполнения;
  - уменьшение сетевого трафика;
- языки реализации:
  - функции CLR (предоставляют значительный выигрыш в производительности для вычислительных задач, работы со строками и бизнес-логикой, позволяют реализовать статистические функции);
  - функции Transact-SQL (лучше приспособлены для логики доступа к данным);
- ограничение использования функций: инструкции внутри функции могут изменять только локальные по отношению к этой функции объекты, например локальные курсоры или переменные.

Компоненты пользовательской функции

- функция принимает ноль и более параметров и возвращает:
  - скалярное значение;
  - таблицу;
- любая пользовательская функция состоит из двух частей:
  - заголовка, содержащего:
    - имя функции с необязательным именем схемы или владельца;
    - имя и тип данных входного параметра;
    - тип данных возвращаемого значения и необязательное имя;
  - тела, состоящего из:
    - одной или нескольких инструкций Transact-SQL, реализующих логику функции;
    - ссылки на сборку .NET .

Определяемые пользователем функции: пример

```
            -- function name
CREATE FUNCTION dbo.GetWeekDay
            -- parameter name and data type
   (@Date datetime)
            -- return value data type
   RETURNS int
            -- begin body definition
   AS BEGIN
            -- performs the action
       RETURN DATEPART (weekday, @Date)
END;
GO

SELECT
   dbo.GetWeekDay(CONVERT(DATETIME,'20020201',101))
   AS DayOfWeek;
```

Виды определяемых пользователем функций

- скалярные функции:
  - возвращают единственное значение скалярного типа;
  - могут использоваться везде, где допустимы выражения возвращаемого типа;
- табличные функции:
  - возвращают значение типа table;
  - могут использоваться везде, где допустимы табличные выражения (в том числе, как альтернатива представлений и хранимых процедур, возвращающих один результирующий набор).

Свойства функций

- свойства пользовательских функций определяют способность ядра СУБД индексировать результаты функции как с помощью индексов вычисляемых столбцов, вызывающих функцию, так и с помощью индексированных представлений, которые на нее ссылаются:
  - детерминизм: способность возвращать один и тот же результат, если предоставлять им один и тот же набор входных значений и использовать одно и то же состояние базы данных;
  - точность: отсутствие операций над числами с плавающей запятой;
  - доступ к данным: осуществление доступа к локальному серверу баз данных с помощью внутрипроцессного управляемого поставщика SQL Server;
  - доступ к системным данным: осуществление доступа к системным метаданным на локальном сервере баз данных с помощью внутрипроцессного управляемого поставщика SQL Server
  - свойство IsSystemVerified:определяет, может ли ядро СУБД проверить детерминизм и точность функции.

Скалярные функции: определение и пример

```
CREATE FUNCTION function_name
        ( [ { @parameter_name [AS]
              scalar_parameter_data_type } [ ,...n ] ] )
     RETURNS scalar_return_data_type
     [ AS ]
     BEGIN
        function_body
        RETURN scalar_expression
     END

CREATE FUNCTION dbo.ufnGetStock(@ProductID int)
RETURNS int
AS -- Returns the stock level for the product.
BEGIN
    DECLARE @ret int;
    SELECT @ret = SUM(p.Quantity)
    FROM Production.ProductInventory p
    WHERE p.ProductID = @ProductID AND p.LocationID = '6';
    IF (@ret IS NULL)
    SET @ret = 0
RETURN @ret
END;
GO
```

Табличные функции: определение и пример встроенной (inline) функции

```
CREATE FUNCTION function_name
        ( [ { @parameter_name [AS]
              scalar_parameter_data_type } [ ,...n ] ] )
     RETURNS @return_variable TABLE < table_type_definition >
     [ AS ]
     BEGIN
         function_body
      RETURN
     END
----
    RETURNS TABLE
    [ AS ]
     RETURN [ ( ] select-statement [ ) ]

CREATE FUNCTION Sales.fn_SalesByStore (@storeid int)
RETURNS TABLE
AS
RETURN
(
    SELECT P.ProductID, P.Name, SUM(SD.LineTotal) AS 'YTD Total'
    FROM Production.Product AS P
          JOIN Sales.SalesOrderDetail AS SD ON SD.ProductID = P.ProductID
          JOIN Sales.SalesOrderHeader AS SH
                              ON SH.SalesOrderID = SD.SalesOrderID
    WHERE SH.CustomerID = @storeid
    GROUP BY P.ProductID, P.Name
);
GO
```

Табличные функции: пример функции, состоящего из нескольких инструкций

```
CREATE FUNCTION dbo.fn_FindReports (@InEmpID INTEGER)
RETURNS @retFindReports TABLE
(   EmployeeID int primary key NOT NULL,
    Name nvarchar(255) NOT NULL,
    Title nvarchar(50) NOT NULL,
    EmployeeLevel int NOT NULL,
    Sort nvarchar (255) NOT NULL )
AS
BEGIN
    WITH DirectReports(Name, Title, EmployeeID, EmployeeLevel, Sort) AS
          (SELECT CONVERT(Varchar(255), c.FirstName + ' ' + c.LastName), e.Title,
          e.EmployeeID, 1, CONVERT(Varchar(255), c.FirstName + ' ' + c.LastName)
          FROM HumanResources.Employee AS e
          JOIN Person.Contact AS c ON e.ContactID = c.ContactID
          WHERE e.EmployeeID = @InEmpID
          UNION ALL
          SELECT CONVERT(Varchar(255), REPLICATE ('| ' , EmployeeLevel) +
          c.FirstName + ' ' + c.LastName), e.Title, e.EmployeeID, EmployeeLevel + 1,
          CONVERT (Varchar(255), RTRIM(Sort) + '| ' + FirstName + ' ' + LastName)
          FROM HumanResources.Employee as e
          JOIN Person.Contact AS c ON e.ContactID = c.ContactID
          JOIN DirectReports AS d ON e.ManagerID = d.EmployeeID
          )
    INSERT @retFindReports
    SELECT EmployeeID, Name, Title, EmployeeLevel, Sort
    FROM DirectReports
    RETURN
END;
GO

SELECT EmployeeID, Name, Title, EmployeeLevel FROM dbo.fn_FindReports(109) ORDER BY
    Sort;
```

Лабораторная работа №8. Хранимые процедуры, курсоры и пользовательские функции

1. Создать хранимую процедуру, производящую выборку из некоторой таблицы и возвращающую результат выборки в виде курсора.
2. Модифицировать хранимую процедуру п.1. таким образом, чтобы выборка осуществлялась с формированием столбца, значение которого формируется пользовательской функцией.
3. Создать хранимую процедуру, вызывающую процедуру п.1., осуществляющую прокрутку возвращаемого курсора и выводящую сообщения, сформированные из записей при выполнении условия, заданного еще одной пользовательской функцией.
4. Модифицировать хранимую процедуру п.2. таким образом, чтобы выборка формировалась с помощью табличной функции.

Вопросы к экзамену

- Хранимые процедуры: назначение, типы
- Курсоры: назначение, типы, обработка
- Определяемые пользователем функции: Скалярные функции. Свойства функций и их влияние на эффективность использования функций
- Определяемые пользователем функции: Табличные функции. Сравнение функций и хранимых процедур