**Хранимые процедуры**

**Создание и выполнение процедур**

Нередко операция с данными представляет набор инструкций, которые необходимо выполнить в определенной последовательности. Например, при добавлении данных покупки товара необходимо внести данные в таблицу заказов. Однако перед этим надо проверить, а есть ли покупаемый товар в наличии. Возможно, при этом понадобится проверить еще ряд дополнительных условий. То есть фактически процесс покупки товара охватывает несколько действий, которые должны выполняться в определенной последовательности. И в этом случае более оптимально будет инкапсулировать все эти действия в один объект - **хранимую процедуру** (stored procedure).

То есть по сути хранимые процедуры представляют набор инструкций, которые выполняются как единое целое. Тем самым хранимые процедуры позволяют упростить комплексные операции и вынести их в единый объект. Изменится процесс покупки товара, соответственно достаточно будет изменить код процедуры. То есть процедура также упрощает управление кодом.

Также хранимые процедуры позволяют ограничить доступ к данным в таблицах и тем самым уменьшить вероятность преднамеренных или неосознанных нежелательных действий в отношении этих данных.

И еще один важный аспект - производительность. Хранимые процедуры обычно выполняются быстрее, чем обычные SQL-инструкции. Все потому что код процедур компилируется один раз при первом ее запуске, а затем сохраняется в скомпилированной форме.

Для создания хранимой процедуры применяется команда **CREATE PROCEDURE** или **CREATE PROC**.

Таким образом, хранимая процедура имеет три ключевых особенности: упрощение кода, безопасность и производительность.

Например, пусть в базе данных есть таблица, которая хранит данные о товарах:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE TABLE Products  (      Id INT IDENTITY PRIMARY KEY,      ProductName NVARCHAR(30) NOT NULL,      Manufacturer NVARCHAR(20) NOT NULL,      ProductCount INT DEFAULT 0,      Price MONEY NOT NULL  ); |

Создадим хранимую процедуру для извлечения данных из этой таблицы:

|  |  |
| --- | --- |
| 1  2  3  4  5 | USE productsdb;  GO  CREATE PROCEDURE ProductSummary AS  SELECT ProductName AS Product, Manufacturer, Price  FROM Products |

Поскольку команда CREATE PROCEDURE должна вызываться в отдельном пакете, то после команды USE, которая устанавливает текущую базу данных, используется команда **GO** для определения нового пакета.

После имени процедуры должно идти ключевое слово AS.

Для отделения тела процедуры от остальной части скрипта код процедуры нередко помещается в блок BEGIN...END:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | USE productsdb;  GO  CREATE PROCEDURE ProductSummary AS  BEGIN      SELECT ProductName AS Product, Manufacturer, Price      FROM Products  END; |

После добавления процедуры мы ее можем увидеть в узле базы данных в SQL Server Management Studio в подузле **Programmability -> Stored Procedures**:

И мы сможем управлять процедурой также и через визуальный интерфейс.

**Выполнение процедуры**

Для выполнения хранимой процедуры вызывается команда **EXEC** или **EXECUTE**:

|  |  |
| --- | --- |
| 1 | EXEC ProductSummary |

**Удаление процедуры**

Для удаления процедуры применяется команда **DROP PROCEDURE**:

|  |  |
| --- | --- |
| 1 | DROP PROCEDURE ProductSummary |

**Параметры в процедурах**

Процедуры могут принимать параметры. Параметры бывают входными - с их помощью в процедуру можно передать некоторые значения. И также параметры бывают выходными - они позволяют возвратить из процедуры некоторое значение.

Например, пусть в базе данных будет следующая таблица Products:

|  |  |
| --- | --- |
|  | USE productsdb;  CREATE TABLE Products  (      Id INT IDENTITY PRIMARY KEY,      ProductName NVARCHAR(30) NOT NULL,      Manufacturer NVARCHAR(20) NOT NULL,      ProductCount INT DEFAULT 0,      Price MONEY NOT NULL  ); |

Определим процедуру, которая будет добавлять данные в эту таблицу:

|  |  |
| --- | --- |
|  | USE productsdb;  GO  CREATE PROCEDURE AddProduct      @name NVARCHAR(20),      @manufacturer NVARCHAR(20),      @count INT,      @price MONEY  AS  INSERT INTO Products(ProductName, Manufacturer, ProductCount, Price)  VALUES(@name, @manufacturer, @count, @price) |

После названия процедуры идет список входных параметров, которые определяются также как и переменные - название начинается с символа **@**, а после названия идет тип переменной. И с помощью команды INSERT значения этих параметров будут передаваться в таблицу Products.

Используем эту процедуру:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | USE productsdb;    DECLARE @prodName NVARCHAR(20), @company NVARCHAR(20);  DECLARE @prodCount INT, @price MONEY  SET @prodName = 'Galaxy C7'  SET @company = 'Samsung'  SET @price = 22000  SET @prodCount = 5    EXEC AddProduct @prodName, @company, @prodCount, @price    SELECT \* FROM Products |

Здесь передаваемые в процедуру значения определяются через переменные. При вызове процедуры ей через запятую передаются значения. При этом значения передаются параметрам процедуры **по позиции**. Так как первым определен параметр @name, то ему будет передаваться первое значение - значение переменной @prodName. Второму параметру - @manufacturer передается второе значение - значение переменной @company и так далее. Главное, чтобы между передаваемыми значениями и параметрами процедуры было соответствие по типу данных.

Также можно было бы передать непосредственно значения:

|  |  |
| --- | --- |
| 1 | EXEC AddProduct 'Galaxy C7', 'Samsung', 5, 22000 |

Также значения параметрам процедуры можно передавать по имени:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | USE productsdb;    DECLARE @prodName NVARCHAR(20), @company NVARCHAR(20);  SET @prodName = 'Honor 9'  SET @company = 'Huawei'    EXEC AddProduct @name = @prodName,                  @manufacturer=@company,                  @count = 3,                  @price = 18000 |

При передаче параметров по имени параметру процедуры присваивается некоторое значение.

**Необязательные параметры**

Параметры можно отмечать как необязательные, присваивая им некоторое значение по умолчанию. Например, в случае выше мы можем автоматически устанавливать для количества товара значение 1, если соответствующее значение не передано в процедуру:

|  |  |
| --- | --- |
|  | USE productsdb;  GO  CREATE PROCEDURE AddProductWithOptionalCount      @name NVARCHAR(20),      @manufacturer NVARCHAR(20),      @price MONEY,      @count INT = 1  AS  INSERT INTO Products(ProductName, Manufacturer, ProductCount, Price)  VALUES(@name, @manufacturer, @count, @price) |

При этом необязательные параметры лучше помещать в конце списка параметров процедуры.

|  |  |
| --- | --- |
|  | DECLARE @prodName NVARCHAR(20), @company NVARCHAR(20), @price MONEY  SET @prodName = 'Redmi Note 5A'  SET @company = 'Xiaomi'  SET @price = 22000    EXEC AddProductWithOptionalCount @prodName, @company, @price    SELECT \* FROM Products |

И в этом случае для параметра @count в процедуру можно не передавать значение.

**Выходные параметры и возвращение результата**

Выходные параметры позволяют возвратить из процедуры некоторый результат. Выходные параметры определяются с помощью ключевого слова **OUTPUT**.

Например, определим еще одну процедуру:

|  |  |
| --- | --- |
|  | USE productsdb;  GO  CREATE PROCEDURE GetPriceStats      @minPrice MONEY OUTPUT,      @maxPrice MONEY OUTPUT  AS  SELECT @minPrice = MIN(Price),  @maxPrice = MAX(Price)  FROM Products |

При вызове процедуры для выходных параметров передаются переменные с ключевым словом OUTPUT:

|  |  |
| --- | --- |
|  | USE productsdb;  DECLARE @minPrice MONEY, @maxPrice MONEY    EXEC GetPriceStats @minPrice OUTPUT, @maxPrice OUTPUT    PRINT 'Минимальная цена ' + CONVERT(VARCHAR, @minPrice)  PRINT 'Максимальная цена ' + CONVERT(VARCHAR, @maxPrice) |

Также можно сочетать входные и выходные параметры. Например, определим процедуру, которая добавляет новую строку в таблицу и возвращает ее id:

|  |  |
| --- | --- |
|  | USE productsdb;  GO    CREATE PROCEDURE CreateProduct      @name NVARCHAR(20),      @manufacturer NVARCHAR(20),      @count INT,      @price MONEY,      @id INT OUTPUT  AS      INSERT INTO Products(ProductName, Manufacturer, ProductCount, Price)      VALUES(@name, @manufacturer, @count, @price)      SET @id = @@IDENTITY |

С помощью глобальной переменной **@@IDENTITY** можно получить идентификатор добавленной записи.

При вызове этой процедуры ей также по позиции передаются все входные и выходные параметры:

|  |  |
| --- | --- |
|  | USE productsdb;    DECLARE @id INT    EXEC CreateProduct 'LG V30', 'LG', 3, 28000, @id OUTPUT    PRINT @id |

**Возвращение значения**

Кроме передачи результата выполнения через выходные параметры хранимая процедура также может возвращать какое-либо значение типа **INT** с помощью оператора **RETURN**. Хотя данная возможность во многом нивелирована использованием выходных параметров, через которые можно возвращать результат, тем не менее, если надо возвратить из процедуры одно значение, то вполне можно использовать оператор RETURN.

Например, возвратим среднюю цену на товары:

|  |  |
| --- | --- |
|  | USE productsdb;  GO  CREATE PROCEDURE GetAvgPrice AS  DECLARE @avgPrice MONEY  SELECT @avgPrice = AVG(Price)  FROM Products  RETURN @avgPrice; |

После оператора RETURN указывается возвращаемое значение. В данном случае это значение переменной @avgPrice.

Вызовем данную процедуру:

|  |  |
| --- | --- |
|  | USE productsdb;    DECLARE @result MONEY    EXEC @result = GetAvgPrice  PRINT @result |

Для получения результата процедуры ее значение сохраняется в переменную (в данном случае в переменную @result):

Стоит отметить, что RETURN возвращает только целочисленные значения.

**Триггеры**

**Определение триггеров**

Триггеры представляют специальный тип хранимой процедуры, которая вызывается автоматически при выполнении определенного действия над таблицей или представлением, в частности, при добавлении, изменении или удалении данных, то есть при выполнении команд INSERT, UPDATE, DELETE.

Формальное определение триггера:

|  |  |
| --- | --- |
|  | CREATE TRIGGER имя\_триггера  ON {имя\_таблицы | имя\_представления}  {AFTER | INSTEAD OF} [INSERT | UPDATE | DELETE]  AS выражения\_sql |

Для создания триггера применяется выражение **CREATE TRIGGER**, после которого идет имя триггера. Как правило, имя триггера отражает тип операций и имя таблицы, над которой производится операция.

Каждый триггер ассоциируется с определенной таблицей или представлением, имя которых указывается после слова **ON**.

Затем устанавливается тип триггера. Мы можем использовать один из двух типов:

* **AFTER**: выполняется после выполнения действия. Определяется только для таблиц.
* **INSTEAD OF**: выполняется вместо действия (то есть по сути действие - добавление, изменение или удаление - вообще не выполняется). Определяется для таблиц и представлений

После типа триггера идет указание операции, для которой определяется триггер: **INSERT**, **UPDATE** или **DELETE**.

Для триггера AFTER можно применять сразу для нескольких действий, например, UPDATE и INSERT. В этом случае операции указываются через запятую. Для триггера INSTEAD OF можно определить только одно действие.

И затем после слова AS идет набор выражений SQL, которые собственно и составляют тело триггера.

Создадим триггер. Допустим, у нас есть база данных productsdb со следующим определением:

|  |  |
| --- | --- |
|  | CREATE DATABASE productdb;  GO    USE productdb;  CREATE TABLE Products  (      Id INT IDENTITY PRIMARY KEY,      ProductName NVARCHAR(30) NOT NULL,      Manufacturer NVARCHAR(20) NOT NULL,      ProductCount INT DEFAULT 0,      Price MONEY NOT NULL  ); |

Определим триггер, который будет срабатывать при добавлении и обновлении данных:

|  |  |
| --- | --- |
|  | USE productdb;  GO  CREATE TRIGGER Products\_INSERT\_UPDATE  ON Products  AFTER INSERT, UPDATE  AS  UPDATE Products  SET Price = Price + Price \* 0.38  WHERE Id = (SELECT Id FROM inserted) |

Допустим, в таблице Products хранятся данные о товарах. Но цена товара нередко содержит различные надбавки типа налога на добавленную стоимость, налога на добавленную коррупцию и так далее. Человек, добавляющий данные, может не знать все эти тонкости с налоговой базой, и он определяет чистую цену. С помощью триггера мы можем поправить цену товара на некоторую величину.

Таким образом, триггер будет срабатывать при любой операции INSERT или UPDATE над таблицей Products. Сам триггер будет изменять цену товара, а для получения того товара, который был добавлен или изменен, находим этот товар по Id. Но какое значение должен иметь Id такой товар? Дело в том, что при добавлении или изменении данные сохраняются в промежуточную таблицу inserted. Она создается автоматически. И из нее мы можем получить данные о добавленных/измененных товарах.

И после добавления товара в таблицу Products в реальности товар будет иметь несколько большую цену, чем та, которая была определена при добавлении:

**Удаление триггера**

Для удаления триггера необходимо применить команду **DROP TRIGGER**:

|  |  |
| --- | --- |
| 1 | DROP TRIGGER Products\_INSERT\_UPDATE |

**Отключение триггера**

Бывает, что мы хотим приостановить действие триггера, но удалять его полностью не хотим. В этом случае его можно временно отключить с помощью команды **DISABLE TRIGGER**:

|  |  |
| --- | --- |
| 1 | DISABLE TRIGGER Products\_INSERT\_UPDATE ON Products |

А когда триггер понадобится, его можно включить с помощью команды **ENABLE TRIGGER**:

|  |  |
| --- | --- |
| 1 | ENABLE TRIGGER Products\_INSERT\_UPDATE ON Products |

**Триггеры для операций INSERT, UPDATE, DELETE**

Для рассмотрения операций с триггерами определим следующую базу данных productsdb:

|  |  |
| --- | --- |
|  | CREATE DATABASE productsdb;  GO  USE productsdb;  CREATE TABLE Products  (      Id INT IDENTITY PRIMARY KEY,      ProductName NVARCHAR(30) NOT NULL,      Manufacturer NVARCHAR(20) NOT NULL,      ProductCount INT DEFAULT 0,      Price MONEY NOT NULL  );  CREATE TABLE History  (      Id INT IDENTITY PRIMARY KEY,      ProductId INT NOT NULL,      Operation NVARCHAR(200) NOT NULL,      CreateAt DATETIME NOT NULL DEFAULT GETDATE(),  ); |

Здесь определены две таблиц: Products - для хранения товаров и History - для хранения истории операций с товарами.

**Добавление**

При добавлении данных (при выполнении команды **INSERT**) в триггере мы можем получить добавленные данные из виртуальной таблицы **INSERTED**.

Определим триггер, который будет срабатывать после добавления:

|  |  |
| --- | --- |
|  | USE productsdb  GO  CREATE TRIGGER Products\_INSERT  ON Products  AFTER INSERT  AS  INSERT INTO History (ProductId, Operation)  SELECT Id, 'Добавлен товар ' + ProductName + '   фирма ' + Manufacturer  FROM INSERTED |

Этот триггер будет добавлять в таблицу History данные о добавлении товара, которые берутся из виртуальной таблицы INSERTED.

Выполним добавление данных в Products и получим данные из таблицы History:

|  |
| --- |
| USE productsdb;  INSERT INTO Products (ProductName, Manufacturer, ProductCount, Price)  VALUES('iPhone X', 'Apple', 2, 79900)    SELECT \* FROM History |

**Удаление данных**

При удалении все удаленные данные помещаются в виртуальную таблицу **DELETED**:

|  |  |
| --- | --- |
|  | USE productsdb  GO  CREATE TRIGGER Products\_DELETE  ON Products  AFTER DELETE  AS  INSERT INTO History (ProductId, Operation)  SELECT Id, 'Удален товар ' + ProductName + '   фирма ' + Manufacturer  FROM DELETED |

Здесь, как и в случае с предыдущим триггером, помещаем информацию об удаленных товарах в таблицу History.

Выполним команду на удаление:

|  |  |
| --- | --- |
|  | USE productsdb;  DELETE FROM Products  WHERE Id=2    SELECT \* FROM History |

**Изменение данных**

Триггер обновления данных срабатывает при выполнении операции UPDATE. И в таком триггере мы можем использовать две виртуальных таблицы. Таблица INSERTED хранит значения строк после обновления, а таблица DELETED хранит те же строки, но до обновления.

Создадим триггер обновления:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | USE productsdb  GO  CREATE TRIGGER Products\_UPDATE  ON Products  AFTER UPDATE  AS  INSERT INTO History (ProductId, Operation)  SELECT Id, 'Обновлен товар ' + ProductName + '   фирма ' + Manufacturer  FROM INSERTED |

И при обновлении данных сработает данный триггер:

## Триггер INSTEAD OF

Триггер INSTEAD OF срабатывает вместо операции с данными. Он определяется в принципе также, как триггер AFTER, за тем исключением, что он может определяться только для одной операции - INSERT, DELETE или UPDATE. И также он может применяться как для таблиц, так и для представлений (триггер AFTER применяется только для таблиц).

Например, создадим следующие базу данных и таблицу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | CREATE DATABASE prods;  GO  USE prods;  CREATE TABLE Products  (      Id INT IDENTITY PRIMARY KEY,      ProductName NVARCHAR(30) NOT NULL,      Manufacturer NVARCHAR(20) NOT NULL,      Price MONEY NOT NULL,      IsDeleted BIT NULL  ); |

Здесь таблица содержит столбец IsDeleted, который указывает, удалена ли запись. То есть вместо жесткого удаления полностью из базы данных мы хотим выполнить мягкое удаление, при котором запись остается в базе данных.

Определим триггер для удаления записи:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | USE prods  GO  CREATE TRIGGER products\_delete  ON Products  INSTEAD OF DELETE  AS  UPDATE Products  SET IsDeleted = 1  WHERE ID =(SELECT Id FROM deleted) |

Добавим некоторые данные в таблицу и выполним удаление из нее:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | USE prods;    INSERT INTO Products(ProductName, Manufacturer, Price)  VALUES ('iPhone X', 'Apple', 79000),  ('Pixel 2', 'Google', 60000);    DELETE FROM Products  WHERE ProductName='Pixel 2';    SELECT \* FROM Products; |

Таким образом, удаляемые записи на самом деле не будут удаляться, просто у них будет устанавливаться значение для столбца IsDeleted: