

Microsoft®



SQL Server 2012 自習書シリーズ No.6

開発者のための Transact-SQL 応用

Published: 2008 年 4 月 30 日

SQL Server 2012 更新版: 2012 年 9 月 30 日

有限会社エスキューエル・クオリティ



この文章に含まれる情報は、公表の日付の時点での Microsoft Corporation の考え方を表しています。市場の変化に応える必要があるため、Microsoft は記載されている内容を約束しているわけではありません。この文書の内容は印刷後も正しいとは保障できません。この文章は情報の提供のみを目的としています。

Microsoft、SQL Server、Visual Studio、Windows、Windows XP、Windows Server、Windows Vista は Microsoft Corporation の米国およびその他の国における登録商標です。

その他、記載されている会社名および製品名は、各社の商標または登録商標です。

© Copyright 2012 Microsoft Corporation. All rights reserved.

目次

| | |
|---------------------------------------|----|
| STEP 1. 本自習書の概要と 自習書を試す環境について | 5 |
| 1.1 本自習書の内容について..... | 6 |
| 1.2 自習書を試す環境について | 7 |
| 1.3 事前作業 (sampleDB データベースの作成) | 8 |
| STEP 2. 応用的な T-SQL..... | 10 |
| 2.1 SELECT ステートメントの結果をローカル変数へ代入..... | 11 |
| 2.2 動的 SQL | 13 |
| 2.3 sp_executesql | 17 |
| 2.4 TOP 句での変数..... | 22 |
| 2.5 MERGE (UPSERT) | 24 |
| 2.6 ROW_NUMBER、RANK、DENSE_RANK | 27 |
| 2.7 n件目から m件目の取得 (ページング) | 30 |
| 2.8 OFFSET .. FETCH (ページング) | 32 |
| 2.9 一時テーブルによる結果の一時的な保存 | 33 |
| 2.10 テーブル変数..... | 35 |
| 2.11 ユーザー定義テーブル型..... | 37 |
| 2.12 CTE (共通テーブル式) | 39 |
| 2.13 再帰クエリ (CTE) | 41 |
| 2.14 HierarchyID データ型..... | 43 |
| STEP 3. ストアド プロシージャ | 44 |
| 3.1 ストアド プロシージャとは | 45 |
| 3.2 ストアド プロシージャの作成と実行..... | 46 |
| 3.3 入力パラメーター | 48 |
| 3.4 IN 演算子のパラメーター化..... | 53 |
| 3.5 テーブル値パラメーターとユーザー定義テーブル型 | 55 |
| 3.6 出力パラメーター (OUTPUT) | 58 |
| 3.7 出力パラメーターで IDENTITY 値の取得 | 60 |
| 3.8 RETURN コード | 64 |
| 3.9 ストアド プロシージャの削除 | 66 |
| 3.10 ストアド プロシージャの定義の表示..... | 67 |
| STEP 4. トランザクションとエラー処理..... | 69 |
| 4.1 トランザクションとは | 70 |
| 4.2 制約違反エラー時の動作..... | 74 |
| 4.3 SET XACT_ABORT ON の追加 | 76 |
| 4.4 例外処理 : TRY ~ CATCH | 78 |
| 4.5 エラー メッセージの取得 : ERROR_MESSAGE..... | 80 |
| 4.6 エラーの再スロー : THROW | 82 |
| 4.7 ユーザー定義エラー (RAISERROR) | 84 |

| | |
|---|-----|
| STEP 5. その他..... | 91 |
| 5.1 オブジェクトの依存関係の表示 | 92 |
| 5.2 Spatial データ型による地図データのサポート | 95 |
| 5.3 FileTable による Windows ファイルのサポート | 107 |

STEP 1. 本自習書の概要と 自習書を試す環境について

この STEP では、本自習書の概要と自習書を試す環境について説明します。

この STEP では、次のことを学習します。

- ✓ 本自習書の内容について
- ✓ 自習書を試す環境について
- ✓ 事前作業（sampleDB データベースの作成）

1.1 本自習書の内容について

✦ 本自習書の内容について

本自習書では、Transact-SQL (T-SQL) ステートメントの応用的な利用方法を説明します。基本構文については、本自習書シリーズの「**Transact-SQL 入門**」編で説明しています。

Transact-SQL 入門編で説明した内容

- **Transact-SQL の構成要素**（ローカル変数、バッチ、文末、PRINT など）
- **流れ制御構文**（IF、IF EXISTS、WHILE、CASE など）
- **照合順序**（Japanese_CI_AS の動作など）
- **データ型**（char、nchar、int、decimal、datetime など）
- **関数**（日付関数、変換関数、文字列関数、数値関数、ユーザー定義関数など）

本自習書では、以下の内容を説明します。

- **動的 SQL**（EXEC、sp_executesql による SQL の組み立て）
- **応用的な SQL 実行**（MERGE、ROW_NUMBER、一時テーブル、テーブル変数、CTE、再帰クエリ、HirerarchID など）
- **ストアド プロシージャ**（入力／出力パラメーター、RETURN コードなど）
- **ユーザー定義テーブル型とテーブル値パラメーター**
- **トランザクション**（BEGIN TRANSACTION、COMMIT TRANSACTION など）
- **エラー処理**（TRY~CATCH、RAISERROR など）
- **オブジェクトの依存関係表示**
- **特殊データ型**（Spatial データ型、FileTable など）

1.2 自習書を試す環境について

✦ 必要な環境

この自習書で実習を行うために必要な環境は、次のとおりです。

OS

Windows Server 2008 SP2 以降 または
Windows Server 2008 R2 SP1 以降 または
Windows Server 2012 または
Windows Vista SP2 以降 または Windows 7 SP1 以降 または Windows 8

ソフトウェア

SQL Server 2012

この自習書内での画面やテキストは、OS に Windows Server 2008 R2 (x64) SP1、ソフトウェアに SQL Server 2012 Enterprise エディション (x64) を利用して記述しています。

そのほか

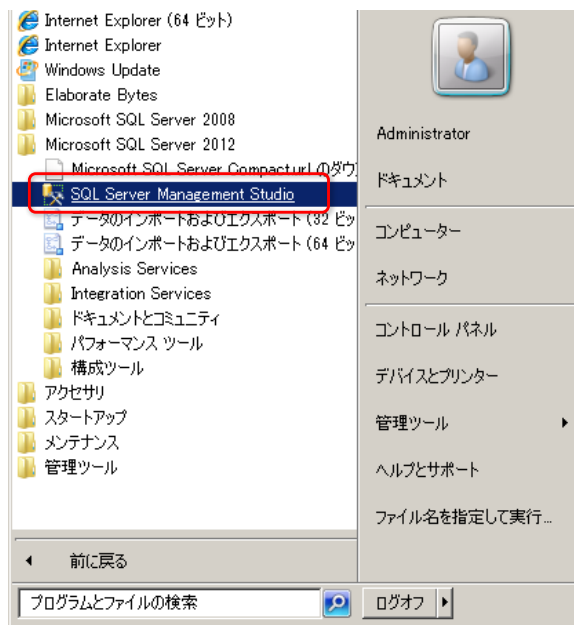
この自習書を試すには、サンプル スクリプトをダウンロードして、次のページの事前作業を実行しておく必要があります。

1.3 事前作業 (sampleDB データベースの作成)

◆ 事前作業

この自習書を進めるには、サンプル スクリプトをダウンロードしておく必要があります。また、Management Studio のクエリ エディターを利用して、サンプル スクリプト内にある「CreateTableEmp.txt」を実行して、「sampleDB」データベースと「emp」テーブルを作成しておく必要があります（実行手順は、次のとおりです）。

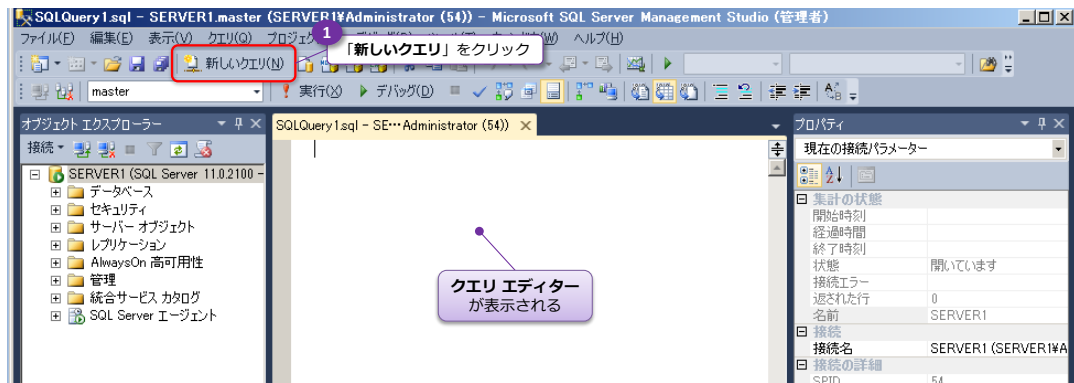
1. まずは、Management Studio を起動するために、[スタート] メニューの [すべてのプログラム] から、[Microsoft SQL Server 2012] を選択して、[SQL Server Management Studio] をクリックします。



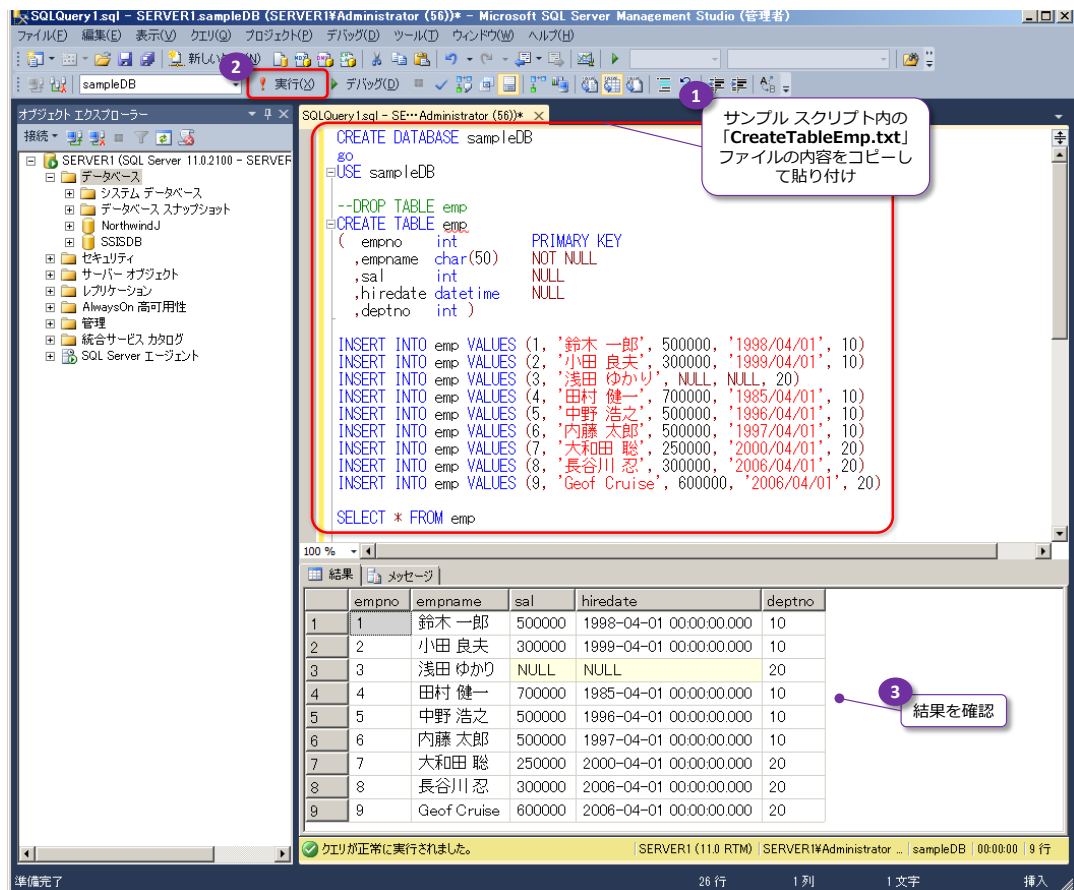
2. 起動後、次のように [サーバーへの接続] ダイアログが表示されたら、[サーバー名] へ SQL Server の名前を入力し、[接続] ボタンをクリックします。



3. 接続完了後、Management Studio が開いたら、次のようにツールバーの [新しいクエリ] ボタンをクリックして、クエリ エディターを開きます。



4. 次に、**Windows エクスプローラー**を起動して、**サンプル スクリプト**をダウンロードしたフォルダーを展開し、このフォルダー内の「**CreateTableEmp.txt**」ファイルをダブル クリックして開きます。ファイルの内容をすべてコピーして、クエリ エディターへ貼り付けます。



貼り付け後、ツールバーの「**実行**」ボタンをクリックしてクエリを実行します。これにより、「**sampleDB**」データベースが作成され、その中へ「**emp**」テーブルが作成されます。実行後、「**emp**」テーブルの **9 件**のデータが表示されれば、実行が完了です。

STEP 2. 応用的な T-SQL

この STEP では、応用的な Transact-SQL ステートメントの利用方法を説明します。「SELECT ステートメントの結果を変数へ代入する方法」や「動的 SQL」、「MERGE ステートメント」、「ROW_NUMBER」、「ペー징ング」、「一時テーブル」、「テーブル変数」、「CTE」（共通テーブル式）などを説明します。

この STEP では、次のことを学習します。

- ✓ SELECT ステートメントの結果をローカル変数へ代入
- ✓ 動的 SQL による SQL の組み立て
- ✓ TOP 句での変数
- ✓ MERGE ステートメント
- ✓ ROW_NUMBER、RANK、DENSE_RANK
- ✓ n 件目から m 件目までの取得（ペーjing）
- ✓ OFFSET .. FETCH（ペーjing）
- ✓ 一時テーブル
- ✓ テーブル変数
- ✓ CTE（共通テーブル式）
- ✓ 再帰クエリ
- ✓ HierarchyID

2.1 SELECT ステートメントの結果をローカル変数へ代入

SELECT ステートメントの結果をローカル変数へ代入

Transact-SQL のローカル変数へは、SELECT ステートメントで取得した列の値を代入することもできます。これは次のように利用します。

```
DECLARE @変数名1 データ型, @変数名2 データ型, ...
SELECT @変数名1 = 列名1, @変数名2 = 列名2, ... FROM テーブル名 WHERE 条件式
```

Let's Try

それでは、これを試してみましょう。

1. まずは、**Management Studio** の [**クエリ エディター**] を開きます。
2. 次に、Step1 で作成した「**sampleDB**」データベースの「**emp**」テーブルの中身を確認します。

```
USE sampleDB
SELECT * FROM emp
```

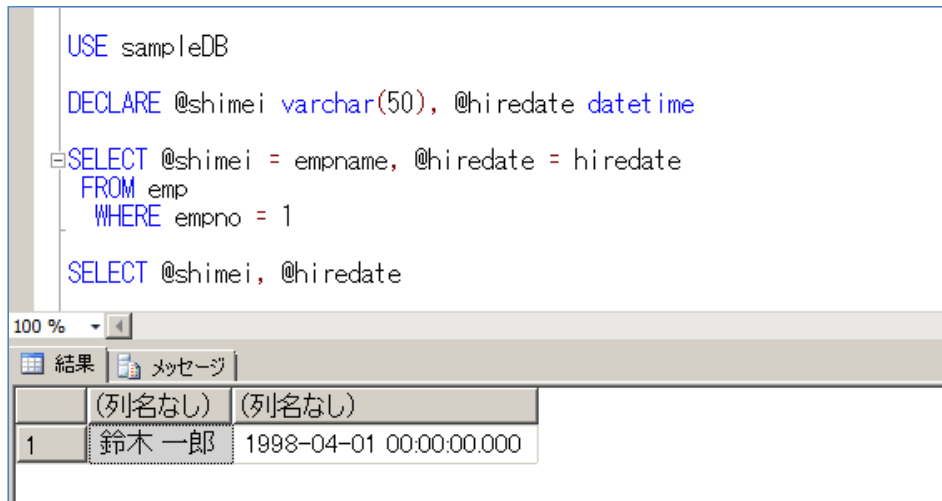
| empno | empname | sal | hiredate | deptno |
|-------|-------------|--------|-------------------------|--------|
| 1 | 鈴木 一郎 | 500000 | 1988-04-01 00:00:00.000 | 10 |
| 2 | 小田 良夫 | 300000 | 1989-04-01 00:00:00.000 | 10 |
| 3 | 浅田 ゆかり | NULL | NULL | 20 |
| 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |
| 5 | 中野 浩之 | 500000 | 1986-04-01 00:00:00.000 | 10 |
| 6 | 内藤 太郎 | 500000 | 1997-04-01 00:00:00.000 | 10 |
| 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |
| 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |

3. 続いて、「**emp**」テーブルに対して、「**empno**」(社員番号) が 1 番の社員の「**empname**」(氏名) と「**hiredate**」(入社日) のデータを取得し、それをローカル変数「**@shimei**」と「**@hiredate**」へ代入してみます。

```
USE sampleDB
DECLARE @shimei varchar(50), @hiredate datetime

SELECT @shimei = empname, @hiredate = hiredate
FROM emp
WHERE empno = 1
```

```
SELECT @shimei, @hiredate
```



このように Transact-SQL では、SELECT ステートメントで取得した結果をローカル変数へ代入することができます。

Note : SELECT ステートメントの結果が 1 件になるように WHERE 句の条件式を指定

SELECT ステートメントで取得した結果をローカル変数へ代入する場合は、SELECT ステートメントの結果が 1 件になるように WHERE 句の条件式を指定する必要があります。ローカル変数へは、(後述するテーブル変数の場合を除いて)、1 つの値しか格納できないからです。もし、複数の結果が返る場合は、最後に取得した結果が変数へ格納されるのですが、SELECT ステートメントでは、(ORDER BY 句を指定しない限り) 結果の順序に保証はないので、そのような利用方法はお勧めしません。

2.2 動的 SQL

✦ 動的 SQL とは

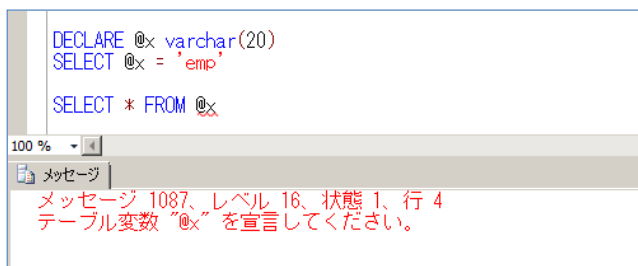
動的 SQL は、動的に（ローカル変数の値に応じて）SQL を組み立てて実行する機能で、「EXECUTE」ステートメントまたは「sp_executesql」システム ストアド プロシージャを利用します。これにより、SQL ステートメントの一部をパラメーター化して実行できるようになります。

✦ テーブル名や列名の変数化（パラメーター化）

SELECT ステートメントなどで、テーブル名や列名を変数化して実行したい場合には、動的 SQL を利用しなければなりません。なぜ、動的 SQL を利用しなければならないかは、次のステートメントを実行してみると理解できると思います。

```
DECLARE @x varchar(20)
SELECT @x = 'emp'

SELECT * FROM @x
```

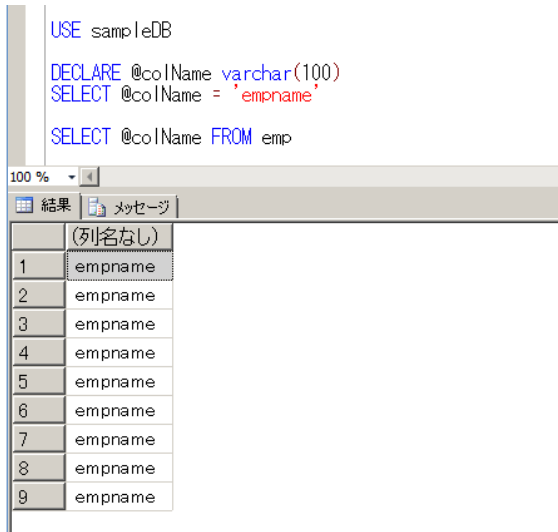


このステートメントは、「SELECT * FROM @x」のように、テーブル名の部分へ変数を利用していますが、結果は、「**テーブル変数 "@x" を宣言してください**」というエラーが発生しています。テーブル変数については後述しますが、FROM の後に記述できるローカル変数は、テーブル変数のみで、通常のローカル変数を指定することができないのです。

また、次のように SELECT ステートメントの列名の部分へ変数を利用しようとしても、同様に正しく結果を取得することができません。

```
USE sampleDB
DECLARE @colName varchar(100)
SELECT @colName = 'empname'

SELECT @colName FROM emp
```



結果は、@colName 変数へ格納された文字列が emp テーブルの結果の件数分出力されています。これは、SELECT ステートメントが次のように解釈されているためです。

```
SELECT 'empname' FROM emp
```

このようにテーブル名や列名を変数化したい場合には、そのままでは利用できないので、「**動的 SQL**」を利用しなければなりません。

➤ EXECUTE ステートメントによる動的 SQL

前述したように、動的 SQL は、「**EXECUTE**」ステートメントまたは「**sp_executesql**」システム ストアド プロシージャを利用して、実行することができます。まずは EXECUTE ステートメントからみていきましょう。

EXECUTE ステートメントは、次のように利用します。

```
EXECUTE ( { '文字列' | ローカル変数 } )
```

EXECUTE の後に、カッコを記述して、実行したい SQL ステートメントの文字列またはローカル変数を指定します。EXECUTE は、「**EXEC**」と省略することも可能です。

➤ Let's Try

それでは、これを試してみましょう。

1. まずは、次のように記述して、単純に EXECUTE ステートメントの引数へ SELECT ステートメントを文字列として指定して実行してみましょう。

```
USE sampleDB
EXECUTE ('SELECT * FROM emp')
```

```
USE sampleDB
EXECUTE ('SELECT * FROM emp')
```

| | empno | empname | sal | hiredate | deptno |
|---|-------|-------------|--------|-------------------------|--------|
| 1 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |
| 2 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 3 | 3 | 浅田 ゆかり | NULL | NULL | 20 |
| 4 | 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |
| 5 | 5 | 中野 浩之 | 500000 | 1996-04-01 00:00:00.000 | 10 |
| 6 | 6 | 内藤 太郎 | 500000 | 1997-04-01 00:00:00.000 | 10 |
| 7 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |
| 8 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 9 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |

2. 次に、EXECUTE を **EXEC** へ省略して実行してみましょう。

```
EXEC ('SELECT * FROM emp')
```

```
EXEC ('SELECT * FROM emp')
```

| | empno | empname | sal | hiredate | deptno |
|---|-------|-------------|--------|-------------------------|--------|
| 1 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |
| 2 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 3 | 3 | 浅田 ゆかり | NULL | NULL | 20 |
| 4 | 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |
| 5 | 5 | 中野 浩之 | 500000 | 1996-04-01 00:00:00.000 | 10 |
| 6 | 6 | 内藤 太郎 | 500000 | 1997-04-01 00:00:00.000 | 10 |
| 7 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |
| 8 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 9 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |

同じ結果を取得できたことを確認できます。

3. 続いて、テーブル名を変数「@x」へ格納して、文字列連結のための演算子「+」を利用して、ステートメントを実行してみましょう。

```
DECLARE @x varchar(20)
SELECT @x = 'emp'

EXEC ('SELECT * FROM ' + @x)
```

```
DECLARE @x varchar(20)
SELECT @x = 'emp'

EXEC ('SELECT * FROM ' + @x)
```

| | empno | empname | sal | hiredate | deptno |
|---|-------|-------------|--------|-------------------------|--------|
| 1 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |
| 2 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 3 | 3 | 浅田 ゆかり | NULL | NULL | 20 |
| 4 | 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |
| 5 | 5 | 中野 浩之 | 500000 | 1996-04-01 00:00:00.000 | 10 |
| 6 | 6 | 内藤 太郎 | 500000 | 1997-04-01 00:00:00.000 | 10 |
| 7 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |
| 8 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 9 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |

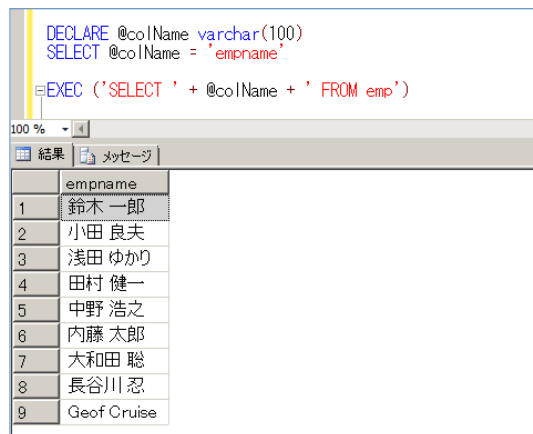
結果は、「SELECT * FROM emp」と同じものが取得できていることを確認できます。このようにテーブル名を変数化（パラメーター化）したい場合には、文字列として組み立てて、

EXECUTE ステートメント（または後述の `sp_executesql`）で動的 SQL として実行するようにします。

✦ 列名の変数化

4. 続いて、次のように列名を変数化して実行してみましょう。

```
DECLARE @colName varchar(100)
SELECT @colName = 'empname'
EXEC ('SELECT ' + @colName + ' FROM emp')
```



```
DECLARE @colName varchar(100)
SELECT @colName = 'empname'
EXEC ('SELECT ' + @colName + ' FROM emp')
```

| | empname |
|---|-------------|
| 1 | 鈴木 一郎 |
| 2 | 小田 良夫 |
| 3 | 浅田 ゆかり |
| 4 | 田村 健一 |
| 5 | 中野 浩之 |
| 6 | 内藤 太郎 |
| 7 | 大和田 聡 |
| 8 | 長谷川 忍 |
| 9 | Geof Cruise |

emp テーブルの empname 列のデータを取得できたことを確認できます。このように列名を変数化したい場合にも、動的 SQL を利用するようにします。

2.3 sp_executesql

➤ sp_executesql による動的 SQL

動的 SQL は、「sp_executesql」システム ストアド プロシージャを利用しても実行することができます。これは次のように利用します。

```
[EXECUTE] sp_executesql N'文字列' | ローカル変数
```

sp_executesql では、N プレフィックスを付けて、文字列またはローカル変数として、実行したい SQL ステートメントを指定します（N プレフィックスについては、本自習書シリーズの「Transact-SQL 入門」編で説明しています）。先頭の EXECUTE は、sp_executesql をバッチの先頭で実行する場合には省略することができます。

➤ Let's Try

それでは、これを試してみましょう。

1. まずは、次のように nvarchar データ型の変数として「@sql」を定義し、文字列 "emp" を格納した変数「@x」と文字列連結して、それを sp_executesql の第 1 引数へ与えてみましょう。

```
USE sampleDB
DECLARE @sql nvarchar(100), @x varchar(10)
SELECT @x = 'emp'
SELECT @sql = N'SELECT * FROM ' + @x

EXECUTE sp_executesql @sql
```

```
USE sampleDB
DECLARE @sql nvarchar(100), @x varchar(10)
SELECT @x = 'emp'
SELECT @sql = N'SELECT * FROM ' + @x

EXECUTE sp_executesql @sql
```

| | empno | empname | sal | hiredate | deptno |
|---|-------|-------------|--------|-------------------------|--------|
| 1 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |
| 2 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 3 | 3 | 浅田 ゆかり | NULL | NULL | 20 |
| 4 | 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |
| 5 | 5 | 中野 浩之 | 500000 | 1996-04-01 00:00:00.000 | 10 |
| 6 | 6 | 内藤 太郎 | 500000 | 1997-04-01 00:00:00.000 | 10 |
| 7 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |
| 8 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 9 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |

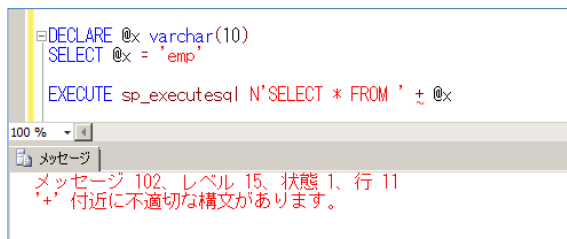
「SELECT * FROM emp」の結果を取得できていることを確認できます。EXECUTE ステートメントとの違いは、完成系の（文字列連結が完了した）SQL ステートメントをローカル変

数として与えている点です。EXECUTE ステートメントでは、次のように引数の中で文字列連結を行うことができました。

```
SELECT @x = 'emp'
EXECUTE ('SELECT * FROM ' + @x)
```

しかし、sp_executesql では、このような記述をした場合は、次のエラーが発生します。

```
SELECT @x = 'emp'
EXECUTE sp_executesql N'SELECT * FROM ' + @x
```



sp_executesql では、文字列連結が完了した SQL ステートメントを引数へ与える必要があることに注意しましょう。

✦ sp_executesql でのパラメーター化

sp_executesql と EXECUTE ステートメントとの一番の違いは、sp_executesql では、SQL のパラメーター化が行える点です。これは、次のように WHERE 句の条件式での値を指定する部分で利用することができます。

```
sp_executesql
N'SELECT .. FROM .. WHERE 列 1 = @パラメーター1 .. 列 2 = @パラメーター2 ...'
, ' @パラメーター1 データ型, @パラメーター2 データ型, ...'
, @パラメーター1 = 代入したい値, @パラメーター2 = 代入したい値, ...
```

ローカル変数と同じように @ を付けてパラメーターを SQL ステートメントの中へ記述し、第 2 引数へパラメーターの定義（データ型の指定）、第 3 引数以降でパラメーターへ代入したい値を指定します。

✦ Let's Try

それでは、これを試してみましょう。

1. ここでは、次の SELECT ステートメントをパラメーター化する場合を考えてみます。

```
SELECT * FROM emp WHERE empname LIKE '%田%' AND sal > 290000
```

```
SELECT * FROM emp WHERE empname LIKE '%田%' AND sal > 290000
```

| | empno | empname | sal | hiredate | deptno |
|---|-------|---------|--------|-------------------------|--------|
| 1 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 2 | 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |

このステートメントは、「**emp**」テーブルに対して、「**empname**」列のデータで“**田**”という文字が含まれていて、かつ「**sal**」（給与）列が“**29 万**”以上のデータへ絞り込んでいます。この検索条件（田や 29 万の部分）は、`sp_executesql` を利用してパラメーター化することができます。

2. 次のように記述して実行してみてください。

```
sp_executesql N'SELECT * FROM emp WHERE empname LIKE @p1 AND sal > @p2'
,N'@p1 varchar(50), @p2 int'
, @p1 = '%田%', @p2 = 290000
```

```
sp_executesql N'SELECT * FROM emp WHERE empname LIKE @p1 AND sal > @p2'
,N'@p1 varchar(50), @p2 int'
, @p1 = '%田%', @p2 = 290000
```

| | empno | empname | sal | hiredate | deptno |
|---|-------|---------|--------|-------------------------|--------|
| 1 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 2 | 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |

前述のクエリと同じ結果を取得できたことを確認できます。`sp_executesql` の第 1 引数では、`@` を付けてパラメーター（`@p1` と `@p2`）を記述し、第 2 引数では“パラメーターに対するデータ型の定義”を行い、第 3 引数以降では“パラメーターへ代入したい値”を指定します。

3. 次に、`@p2` へ与える値を 20 万へ変更して実行してみましょう

```
sp_executesql N'SELECT * FROM emp WHERE empname LIKE @p1 AND sal > @p2'
,N'@p1 varchar(50), @p2 int'
, @p1 = '%田%', @p2 = 200000
```

```
sp_executesql N'SELECT * FROM emp WHERE empname LIKE @p1 AND sal > @p2'
,N'@p1 varchar(50), @p2 int'
, @p1 = '%田%', @p2 = 200000
```

| | empno | empname | sal | hiredate | deptno |
|---|-------|---------|--------|-------------------------|--------|
| 1 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 2 | 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |
| 3 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |

大和田さんを追加で取得できていることを確認できます。

Note : sp_executesql の利点

sp_executesql によるパラメーター化は、実行プランの再利用率を高める効果があるので、パフォーマンスの向上にも貢献します。これは、手順 2 と 手順 3 のように、パラメーターへ与える値が異なる（29 万と 20 万）場合でも、同じ実行プランが利用されて、コンパイル（**クエリ オプティマイザー**による実行プランの選択）の負荷が減るという意味です。実行プランが再利用されたかどうかは、**syscacheobjects** 互換ビューの **usecounts** 列を参照することで確認することができます。

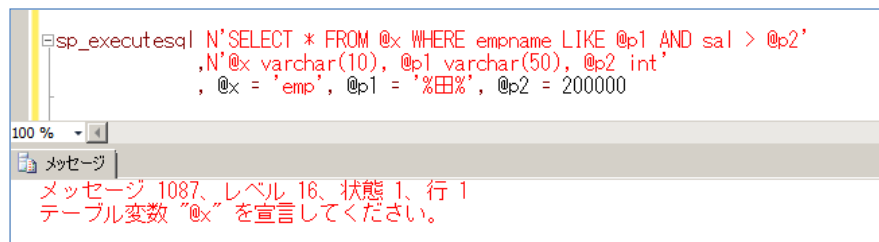
| | sql | usecounts |
|-----|--|-----------|
| 139 | (@p1 varchar(50), @p2 int)SELECT * FROM emp WHERE empname LIKE @p1 AND sal > @p2 | 2 |
| 140 | CREATE PROCEDURE dbo.sp_verify_subsystems @sysssubsystems_refresh_needed BIT = 0 ... | 1 |

また、sp_executesql によるパラメーター化は、SQL インジェクション対策にもなるので、セキュリティの向上にも貢献します。したがって、EXECUTE ステートメントと sp_executesql の利用に悩んだ場合は、積極的に sp_executesql を利用することをお勧めします。パラメーターの指定は、慣れるまでは少々独特の記述になりますが、ぜひ活用してみてください。

➡ テーブル名や列名のパラメーター化

テーブル名や列名などは、sp_executesql のパラメーター機能を利用してパラメーター化することはできません。したがって、次のような実行方法はエラーになります。

```
sp_executesql N'SELECT * FROM @x WHERE empname LIKE @p1 AND sal > @p2'
, N'@x varchar(10), @p1 varchar(50), @p2 int'
, @x = 'emp', @p1 = '%田%', @p2 = 200000
```



あくまでも sp_executesql のパラメーター機能は、WHERE 句の条件式での値を指定する部分のみで利用することができます。したがって、テーブル名や列名をパラメーター化したい場合は、最初に試したように、文字列として組み立てなければなりません（次のように記述します）。

```
DECLARE @sql nvarchar(100), @x varchar(10)
SELECT @x = 'emp'
SELECT @sql = N'SELECT * FROM ' + @x + ' WHERE empname LIKE @p1 AND sal > @p2'

EXEC sp_executesql @sql
, N'@p1 varchar(50), @p2 int'
, @p1 = '%田%', @p2 = 200000
```

```

DECLARE @sql nvarchar(100), @x varchar(10)
SELECT @x = 'emp'
SELECT @sql = N'SELECT * FROM ' + @x + ' WHERE empname LIKE @p1 AND sal > @p2'

EXEC sp_executesql @sql
, N'@p1 varchar(50), @p2 int'
, @p1 = '%田%', @p2 = 200000

```

100 %

結果 メッセージ

| | empno | empname | sal | hiredate | deptno |
|---|-------|---------|--------|-------------------------|--------|
| 1 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 2 | 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |
| 3 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |

Note: SQL インジェクション対策 (テーブル名/列名をパラメーター化する場合はアプリ側で実施)

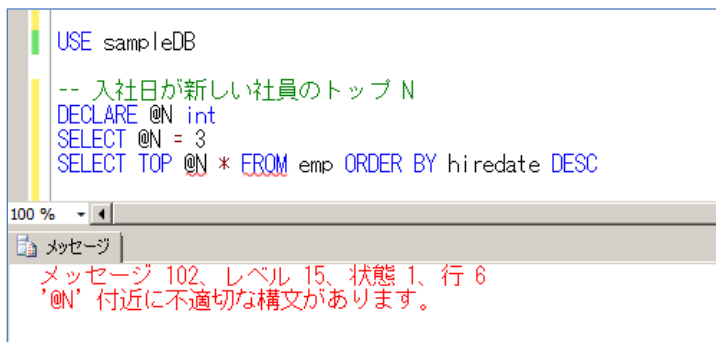
テーブル名や列名をパラメーター化するために、文字列として SQL ステートメントを組み立てた場合は、SQL インジェクション対策にはなりません。この場合は、別途アプリケーション側で対策を施しておく必要があります。

2.4 TOP 句での変数

✦ TOP 句での変数

SQL Server 2000 以前のバージョンでは、次のように TOP 句で変数を記述することはできませんでした。

```
DECLARE @N int
SELECT @N = 3
SELECT TOP @N * FROM emp ORDER BY hiredate DESC
```



これは、SQL Server 2005 以降では解消され、TOP 句を次のようにカッコを付けて実行することで、変数を利用できるようになりました。

```
SELECT TOP (@変数) * FROM ...
```

✦ Let's Try

それでは、これを試してみましょう。

1. 次のように SELECT ステートメントを記述して、入社日が新しい社員のトップ N (N には変数を割り当て) を取得してみましょう。

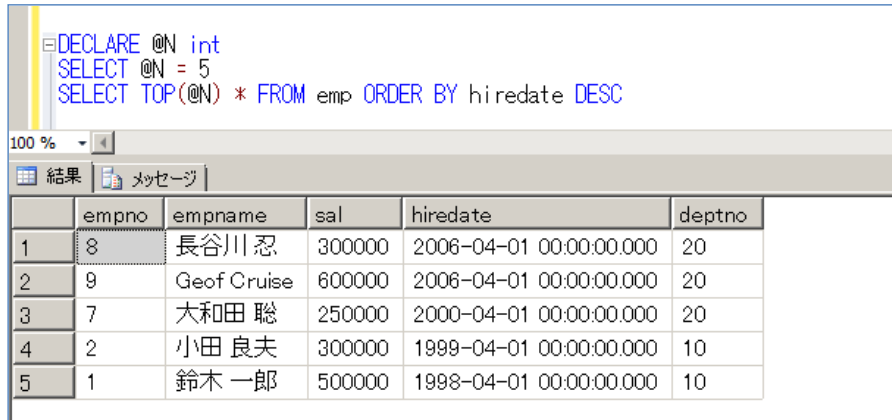
```
DECLARE @N int
SELECT @N = 3
SELECT TOP (@N) * FROM emp ORDER BY hiredate DESC
```

| | empno | empname | sal | hiredate | deptno |
|---|-------|-------------|--------|-------------------------|--------|
| 1 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 2 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |
| 3 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |

TOP 句では、変数をカッコで囲んでいることに注意してください。カッコで囲んで関数のように利用した場合にのみ変数を利用することができます。

2. 次に、@N へ代入する値を 5 へ変更して実行してみましょう。

```
DECLARE @N int
SELECT @N = 5
SELECT TOP (@N) * FROM emp ORDER BY hiredate DESC
```



| | empno | empname | sal | hiredate | deptno |
|---|-------|-------------|--------|-------------------------|--------|
| 1 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 2 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |
| 3 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |
| 4 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 5 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |

今度は 5 件の結果を取得できたことを確認できます。

Note : TOP 句は更新系のステートメントでも利用可能

TOP 句は、DELETE や UPDATE などの更新系のステートメントでも利用することができます。たとえば、DELETE ステートメントの場合は、次のように記述することができます。

```
DECLARE @N = 値
DELETE TOP (@N) FROM t
```

2.5 MERGE (UPSERT)

✦ MERGE ステートメント

MERGE ステートメントは、データが存在する場合には UPDATE、存在しない場合には INSERT 処理が行える非常に便利なステートメントで、SQL Server 2008 から提供された機能です。MERGE ステートメントは、UPDATE と INSERT を組み合わせた造語として「**UPSERT**」とも呼ばれます。Merge は、「結合する」「吸収する」という意味の英単語です。

MERGE ステートメントの構文は、次のとおりです。

```
MERGE INTO マージ先のテーブル
USING マージ元のテーブルまたはクエリ
ON マージの条件
WHEN MATCHED THEN
    UPDATE SET 更新
WHEN NOT MATCHED THEN
    INSERT VALUES ( 追加 );
```

この構文は、Oracle 9i 以降で搭載されている MERGE ステートメントと同じように利用することができます。

✦ Let's Try

それでは、これを試してみましょう。

1. まずは、次のように t1 テーブルと t2 テーブルを作成します。

t1 テーブル

| a | b |
|---|-----|
| 1 | AAA |
| 2 | BBB |
| 3 | CCC |
| 4 | DDD |

t2 テーブル

| a | b |
|---|-----|
| 3 | XXX |
| 5 | YYY |

```
CREATE TABLE t1
( a int, b varchar(100) )
```

```
INSERT INTO t1
VALUES ( 1, 'AAA' )
      , ( 2, 'BBB' )
      , ( 3, 'CCC' )
      , ( 4, 'DDD' )
```

```
SELECT * FROM t1
```

```
CREATE TABLE t2
( a int, b varchar(100) )
```

```
INSERT INTO t2
VALUES ( 3, 'XXX' )
      , ( 5, 'YYY' )
```

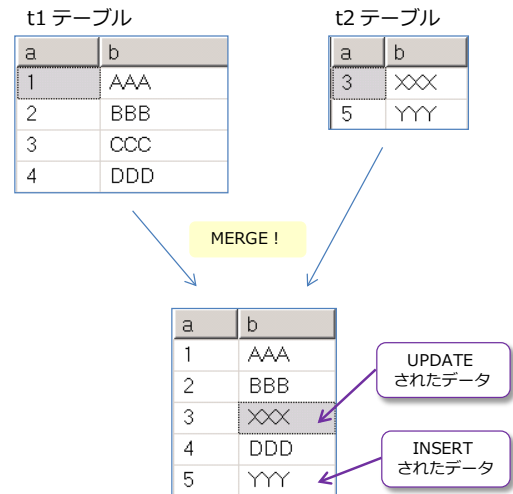
```
SELECT * FROM t2
```


2. 続いて、**t1** テーブルの「**a**」列と、**t2** テーブルの「**a**」列をもとに、次のように **MERGE** ステートメントを実行してみましょう。

```

MERGE INTO t1
USING t2
ON t1.a = t2.a
WHEN MATCHED THEN
    UPDATE SET t1.b = t2.b
WHEN NOT MATCHED THEN
    INSERT VALUES ( t2.a, t2.b );

```



MERGE INTO はマージ（結合）先のテーブルとして「**t1**」、**USING** はマージ対象のテーブルとして「**t2**」を指定し、**ON** はマージの条件（ここでは「**a**」列が等しいかどうか）を指定しています。**WHEN MATCHED**（条件がマッチした場合）には、**THEN** 以下の **UPDATE** ステートメント（更新処理）が実行され、**NOT MATCHED**（マッチしなかった場合）には、その下の **THEN** 以下の **INSERT** ステートメント（挿入処理）が実行されるようになります。

Note : 文末のセミコロンを忘れずに

MERGE ステートメントでは、ステートメントの末尾に必ず「;」（セミコロン）を記述しなければなりません。セミコロンを省略した場合には、エラーになるので注意してください。

✦ 変数をもとにした MERGE

MERGE ステートメントの **USING** には、テーブル名だけでなく、任意のクエリを記述することができます。したがって、複数のテーブル同士の MERGE だけでなく、任意の変数の値をもとにして、MERGE ステートメントを実行することもできます。

1. それでは、これを試してみましょう。次のように変数「**@a**」と「**@b**」を宣言して、これを「**t1**」テーブルとマージしてみます。

```

DECLARE @a int = 4
        ,@b varchar(100) = 'EEE'

```

t1 テーブル

| a | b |
|---|-----|
| 1 | AAA |
| 2 | BBB |
| 3 | XXX |
| 4 | DDD |
| 5 | YYY |

変数（@a と @b）

```

DECLARE @a int = 4
        ,@b varchar(100) = 'EEE'

```

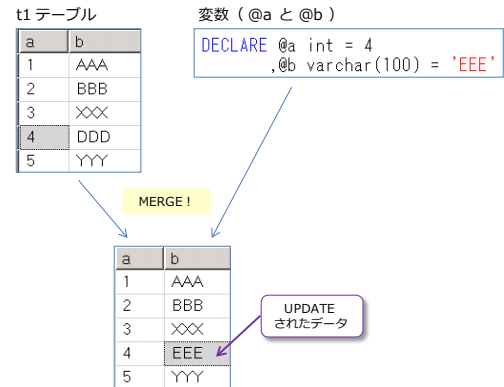
2. MERGE ステートメントは、次のように記述します。

```

DECLARE @a int = 4
        ,@b varchar(100) = 'EEE'

MERGE INTO t1
USING ( SELECT @a AS a, @b AS b ) var
ON t1.a = var.a
WHEN MATCHED THEN
    UPDATE SET t1.b = var.b
WHEN NOT MATCHED THEN
    INSERT VALUES (var.a, var.b );

```



このように USING には、任意のクエリを記述できるので、変数やパラメーターなど特定の値をもとに MERGE を実行することができます。

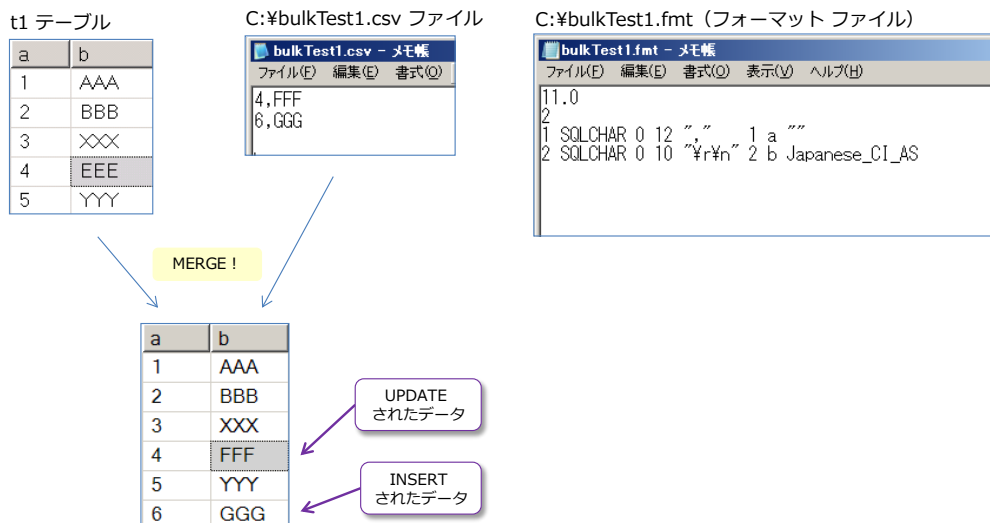
Tips : 一括インポート時に MERGE を利用

MERGE ステートメントでは、OPENROWSET(BULK ...) を指定することもできるので、テキスト ファイルを一括インポートする際に利用することもできます。たとえば、テキスト ファイル (C:\¥bulkTest1.csv) を、フォーマット ファイル (C:\¥bulkTest1.fmt) を利用して、t1 テーブルと MERGE する場合は、次のように記述します。

```

MERGE INTO t1
USING OPENROWSET ( BULK 'C:\¥bulkTest1.csv'
                  ,FORMATFILE = 'C:\¥bulkTest1.fmt' ) bulk1
ON t1.a = bulk1.a
WHEN MATCHED THEN
    UPDATE SET t1.b = bulk1.b
WHEN NOT MATCHED THEN
    INSERT VALUES ( bulk1.a, bulk1.b );

```



フォーマット ファイルについては、本自習書シリーズの「データのコピーと現場で役立つ操作集」で詳しく説明しています。

2.6 ROW_NUMBER、RANK、DENSE_RANK

➤ 順位付け関数

SQL Server では、順位付け関数として「**ROW_NUMBER**」と「**RANK**」、「**DENSE_RANK**」、「**NTILE**」の 4 つが用意されています（これらは SQL Server 2005 から提供されました）。
ROW_NUMBER 関数は、SELECT ステートメントで取得した結果に対して、行番号（結果に対する単純な連番）を取得することができ、RANK と DENSE_RANK、NTILE 関数は、順位（ランク）付けを行うことができる関数です。

構文は、次のとおりです。

関数名 () **OVER** ([PARTITION BY 列名] **ORDER BY** 列名 [DESC])

➤ Let's Try

それでは、これを試してみましょう。

1. まずは、**ROW_NUMBER** 関数を利用して、SELECT ステートメントで取得した結果に対して、行番号を取得してみます。次のように「**emp**」テーブルの「**hiredate**」（入社日）列が新しい順（降順：DESC）に結果を取得します。

```
USE sampleDB
SELECT ROW_NUMBER() OVER (ORDER BY hiredate DESC), * FROM emp
```

| (列名なし) | empno | empname | sal | hiredate | deptno |
|--------|-------|-------------|--------|-------------------------|--------|
| 1 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 2 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |
| 3 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |
| 4 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 5 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |
| 6 | 6 | 内藤 太郎 | 500000 | 1997-04-01 00:00:00.000 | 10 |
| 7 | 5 | 中野 浩之 | 500000 | 1996-04-01 00:00:00.000 | 10 |
| 8 | 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |
| 9 | 3 | 浅田 ゆかり | NULL | NULL | 20 |

➤ RANK、DENSE_RANK、NTILE

1. 次に、RANK と DENSE_RANK、NTILE 関数を利用して、順位を取得してみましょう。
ROW_NUMBER 関数との結果を比較するために、前のクエリとほとんど同じように記述しま

す。

```
SELECT
    ROW_NUMBER() OVER (ORDER BY hiredate DESC)
  , RANK() OVER (ORDER BY hiredate DESC)
  , DENSE_RANK() OVER (ORDER BY hiredate DESC)
  , NTILE(3) OVER (ORDER BY hiredate DESC)
  , * FROM emp
```

| | (列名なし) | (列名なし) | (列名なし) | (列名なし) | empno | empname | sal | hiredate | deptno |
|---|--------|--------|--------|--------|-------|-------------|--------|-------------------------|--------|
| 1 | 1 | 1 | 1 | 1 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 2 | 2 | 1 | 1 | 1 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |
| 3 | 3 | 3 | 2 | 1 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |
| 4 | 4 | 4 | 3 | 2 | 1 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 5 | 5 | 5 | 4 | 2 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |
| 6 | 6 | 6 | 5 | 2 | 6 | 内藤 太郎 | 500000 | 1997-04-01 00:00:00.000 | 10 |
| 7 | 7 | 7 | 6 | 3 | 5 | 中野 浩之 | 500000 | 1996-04-01 00:00:00.000 | 10 |
| 8 | 8 | 8 | 7 | 3 | 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |
| 9 | 9 | 9 | 8 | 3 | 3 | 浅田 ゆかり | NULL | NULL | 20 |

ROW_NUMBER が単純な行番号 (連番) であるのに対して、**RANK** と **DENSE_RANK** 関数は、同じ値があった場合を識別して、同じ順位 (長谷川さんと Geof さんの順位が 1 位タイとなっているなど) を付けることができます。RANK と DENSE_RANK 関数の違いは、同じ値の次の順位を連続にするか、飛ばすかどうかです (大和田さんの順位は、RANK では 3 位、DENSE_RANK では 2 位です)。

NTILE 関数は、引数を与えて利用する必要がありますが、結果を N 等分 (引数が 3 なら 3 等分) して、順位付けをします。したがって、上のクエリ結果は、3 等分されて、長谷川さんから大和田さんまでが 1 位、小田さんから内藤さんまでが 2 位、残りが 3 位となっています。

➤ PARTITION BY 句によるグループ化

順位付け関数は、**PARTITION BY** 句を利用すると、グループ化して、順位を取得することができます。それでは、これを試してみましょう。

1. PARTITION BY 句を利用して、**deptno** (部門番号) でグループ化をして、結果を取得してみましょう。

```
SELECT
    ROW_NUMBER() OVER (PARTITION BY deptno ORDER BY hiredate DESC)
  , RANK() OVER (PARTITION BY deptno ORDER BY hiredate DESC)
  , DENSE_RANK() OVER (PARTITION BY deptno ORDER BY hiredate DESC)
  , NTILE(3) OVER (PARTITION BY deptno ORDER BY hiredate DESC)
  , * FROM emp
```

```

SELECT
  ROW_NUMBER() OVER (PARTITION BY deptno ORDER BY hiredate DESC)
, RANK() OVER (PARTITION BY deptno ORDER BY hiredate DESC)
, DENSE_RANK() OVER (PARTITION BY deptno ORDER BY hiredate DESC)
, NTILE(3) OVER (PARTITION BY deptno ORDER BY hiredate DESC)
FROM emp

```

| | (列名なし) | (列名なし) | (列名なし) | (列名なし) | empno | empname | sal | hiredate | deptno |
|---|--------|--------|--------|--------|-------|-------------|--------|-------------------------|--------|
| 1 | 1 | 1 | 1 | 1 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 2 | 2 | 2 | 2 | 1 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |
| 3 | 3 | 3 | 3 | 2 | 6 | 内藤 太郎 | 500000 | 1997-04-01 00:00:00.000 | 10 |
| 4 | 4 | 4 | 4 | 2 | 5 | 中野 浩之 | 500000 | 1996-04-01 00:00:00.000 | 10 |
| 5 | 5 | 5 | 5 | 3 | 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |
| 6 | 1 | 1 | 1 | 1 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 7 | 2 | 1 | 1 | 1 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |
| 8 | 3 | 3 | 2 | 2 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |
| 9 | 4 | 4 | 3 | 3 | 3 | 浅田 ゆかり | NULL | NULL | 20 |

部門ごとにグループ化

deptno（部門番号）は、10 の社員と 20 の社員がいるので、それぞれの部門ごとに、入社日の新しい順に、順位付けが行われていることを確認できます。

このように、ROW_NUMBER や RANK などの順位関数を利用すると、結果に対して連番や順位を取得できるようになるので、大変便利です。

Tips： ROW_NUMBER 関数の利用しすぎに注意

順位付け関数は、ORDER BY 句を指定していることから分かるように、内部的な並べ替えが伴う処理です。また、PARTITION BY 句を利用した場合は、内部的にはグループ化処理（GROUP BY 演算とほとんど同じ処理）が伴います。これらは、データベース サーバーにとっては、非常に負荷の高い処理（特にメモリとディスクへの高負荷、グループ化で指定する列が多い場合には CPU へも高負荷）なので、使いすぎに注意する必要があります。過去の弊社の案件では、「すべての参照系クエリへ ROW_NUMBER を付けている」というお客様がいらっしゃったのですが、ほとんどのアプリケーション画面で行番号を表示する必要がないにも関わらず ROW_NUMBER を付けているという状態でした。これでは、余計なパフォーマンス ロスが発生しますので、このような使い方はせず、必要な場合にのみ、最低限の場所のみで利用することをお勧めします。

2.7 n 件目から m 件目の取得（ページング）

✦ ROW_NUMBER によるページング

ROW_NUMBER 関数を利用すると、検索結果のうちの「n 件目から m 件目を取得する」といった、いわゆる「ページング」（インターネットの検索エンジン サイトの検索結果などでお馴染みの 10 件ずつデータを表示する機能）を簡単に実現することができます。

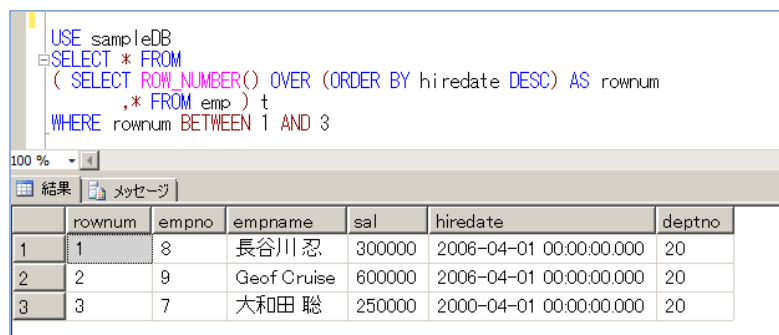
ページングは、ROW_NUMBER で取得した結果（行番号付きの結果）に対して、取り出したい行番号を指定する形で行えます。

✦ Let's Try

それでは、これを試してみましょう。

1. 前の例と同じように、ROW_NUMBER 関数を利用して「emp」テーブルの「hiredate」（入社日）の新しい順に並べ替えた結果を取得し、その結果に対して BETWEEN 演算子で 1 件目から 3 件目を取得してみます。

```
USE sampleDB
SELECT * FROM
( SELECT ROW_NUMBER() OVER (ORDER BY hiredate DESC) AS rownum
  ,* FROM emp ) t
WHERE rownum BETWEEN 1 AND 3
```



The screenshot shows a SQL query window with the following text:

```
USE sampleDB
SELECT * FROM
( SELECT ROW_NUMBER() OVER (ORDER BY hiredate DESC) AS rownum
  ,* FROM emp ) t
WHERE rownum BETWEEN 1 AND 3
```

Below the query window, the 'Results' tab is active, displaying a table with 7 columns: rownum, empno, empname, sal, hiredate, and deptno. The results are as follows:

| | rownum | empno | empname | sal | hiredate | deptno |
|---|--------|-------|-------------|--------|-------------------------|--------|
| 1 | 1 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 2 | 2 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |
| 3 | 3 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |

FROM 句へ（サブクエリとして）カッコ付きで SELECT ステートメントを記述し、その結果に対して「t」という名前を付け、また ROW_NUMBER で取得した行番号へは「rownum」という名前を付けています。そして、WHERE 句で rownum に対して BETWEEN 演算子を指定することで、1 件目から 3 件目（入社日の新しい 3 件のデータ）を取得しています。

2. 次に、BETWEEN の 1 と 3 の部分を 4 と 6 へ置き替えて、4 件目から 6 件目のデータを取得してみましょう。

```
SELECT * FROM
( SELECT ROW_NUMBER() OVER (ORDER BY hiredate DESC) AS rownum
  ,* FROM emp ) t
WHERE rownum BETWEEN 4 AND 6
```

The screenshot shows a SQL query window with the following text:

```

SELECT * FROM
( SELECT ROW_NUMBER() OVER (ORDER BY hiredate DESC) AS rownum
  ,* FROM emp ) t
WHERE rownum BETWEEN 4 AND 6

```

Below the query window, the 'Results' tab is selected, displaying a table with 7 columns: rownum, empno, empname, sal, hiredate, and deptno. The table contains 3 rows of data.

| | rownum | empno | empname | sal | hiredate | deptno |
|---|--------|-------|---------|--------|-------------------------|--------|
| 1 | 4 | 2 | 小田 良夫 | 300000 | 1989-04-01 00:00:00.000 | 10 |
| 2 | 5 | 1 | 鈴木 一郎 | 500000 | 1988-04-01 00:00:00.000 | 10 |
| 3 | 6 | 6 | 内藤 太郎 | 500000 | 1987-04-01 00:00:00.000 | 10 |

このように ROW_NUMBER 関数を利用すると、検索結果に対して、n 件目から m 件目のデータを取得することが簡単に行えるようになります。

Note：インライン ビュー（サブクエリ）

上の例のように FROM 句でカッコを付けて SELECT ステートメントを記述している形のサブクエリ（副問い合わせ）は、「**インライン ビュー**」と呼ばれます。「**ビュー**」は、SELECT ステートメントの結果をテーブルのように扱える（見せかける）ことができる機能ですが、このビューを SELECT ステートメントの内部（インライン）に記述するということから、インライン ビューと呼ばれています。

2.8 OFFSET .. FETCH (ページング)

➤ OFFSET .. FETCH (ページング)

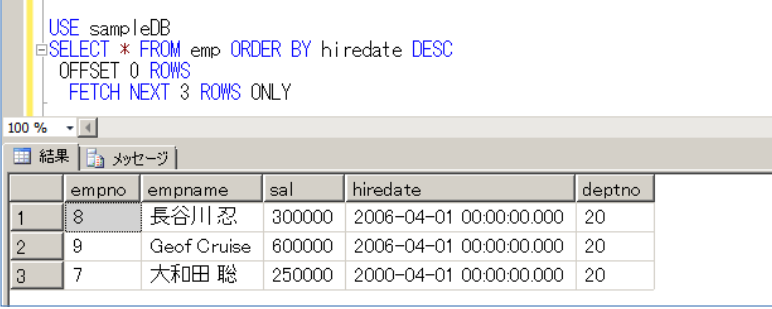
SQL Server 2012 からは、**OFFSET .. FETCH** によるページング機能がサポートされるようになったので、ROW_NUMBER やインライン ビューを利用しなくても、n 件目～ m 件目のデータを取得できるようになりました。

➤ Let's Try

それでは、これを試してみましょう。

1. ROW_NUMBER 関数の例と同じように「**emp**」テーブルの「**hiredate**」(入社日)の新しい順に並べ替えて、1 件目から 3 件目のデータを取得してみます。

```
USE sampleDB
SELECT * FROM emp ORDER BY hiredate DESC
OFFSET 0 ROWS
FETCH NEXT 3 ROWS ONLY
```

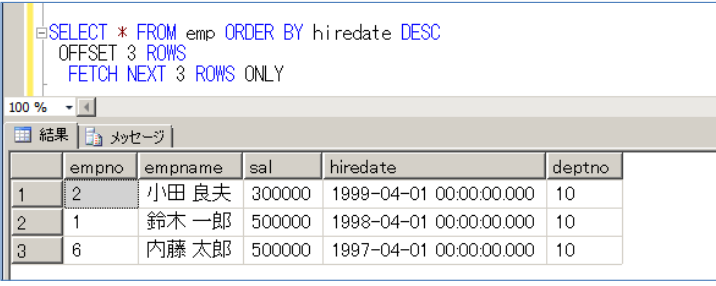


| | empno | empname | sal | hiredate | deptno |
|---|-------|-------------|--------|-------------------------|--------|
| 1 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 2 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |
| 3 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |

OFFSET でスキップしたい件数 (**0 件**) を指定して、**FETCH NEXT** で取得したい件数 (**3 件**) を指定することで、1 件目～ 3 件目のデータを取得することができます。

2. 次に、**OFFSET** の値を **3** に変更して、4 件目から 6 件目のデータを取得してみましょう。

```
SELECT * FROM emp ORDER BY hiredate DESC
OFFSET 3 ROWS
FETCH NEXT 3 ROWS ONLY
```



| | empno | empname | sal | hiredate | deptno |
|---|-------|---------|--------|-------------------------|--------|
| 1 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 2 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |
| 3 | 6 | 内藤 太郎 | 500000 | 1997-04-01 00:00:00.000 | 10 |

このように **OFFSET .. FETCH** を利用すると、簡単にページングを行うことができます。

2.9 一時テーブルによる結果の一時的な保存

✦ 一時テーブルとは

一時テーブルは、ユーザーが接続している間だけ有効な（一時的な）テーブルです。一時テーブルは、次のようにテーブル名の先頭に「#」を付けるだけで作成することができます。

```
CREATE TABLE #一時テーブル名
( 列名1 データ型
, 列名2 データ型
, ...)
```

このように作成したテーブルは、接続が切れると自動的に削除されますが、DROP TABLE ステートメントによる明示的な削除も可能です。

一時テーブルは、次のように **SELECT INTO** ステートメントを利用して、SELECT ステートメントの検索結果をもとに作成することもできます。

```
SELECT * INTO #一時テーブル名 FROM テーブル名 ...
```

一時テーブルは、インライン ビューや後述のテーブル変数、CTE（共通テーブル式）などの置き換えとしても利用できる大変便利な機能です。

✦ Let's Try

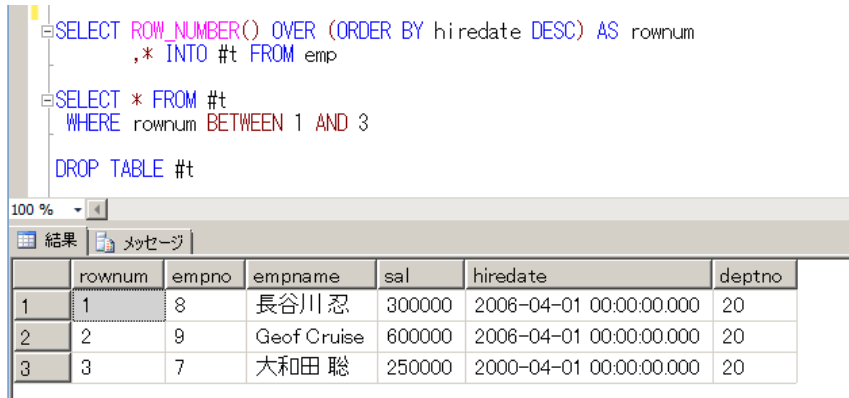
それでは、これを試してみましょう。

1. まずは、ページングのところで利用したインライン ビューを利用した SELECT ステートメントを一時テーブルへ置き替えてみましょう。一時テーブルの作成は、次のように SELECT INTO を利用します。

```
USE sampleDB
SELECT ROW_NUMBER() OVER (ORDER BY hiredate DESC) AS rownum
      , * INTO #t FROM emp

SELECT * FROM #t
WHERE rownum BETWEEN 1 AND 3

DROP TABLE #t
```



```

SELECT ROW_NUMBER() OVER (ORDER BY hiredate DESC) AS rownum
,* INTO #t FROM emp

SELECT * FROM #t
WHERE rownum BETWEEN 1 AND 3

DROP TABLE #t

```

| | rownum | empno | empname | sal | hiredate | deptno |
|---|--------|-------|-------------|--------|-------------------------|--------|
| 1 | 1 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 2 | 2 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |
| 3 | 3 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |

インライン ビューや OFFSET .. FETCH を利用した場合と同じ結果（入社日の新しい社員の 1 件目から 3 件目）を取得できていることを確認できます。

Tips： インライン ビューと一時テーブルの使い分け

インライン ビューと一時テーブルでは、単純なクエリの場合は、インライン ビューのほうがパフォーマンスが良い場合が多いのですが、複雑なクエリ（サブクエリが 5 階層、6 階層と、何段階もの入れ子になって利用されるようなケース）の場合には、一時テーブルを利用したほうが（筆者の経験的には）パフォーマンスが良い場合が多くなります。

インライン ビューも、内部的には、一時的な作業テーブルを作成しているのですが、一時テーブルの場合とほとんど同じ内部処理になるのですが、サブクエリが何段階もの入れ子になっている場合は、クエリを解析して実行プラン（内部的な実行方法）を選択する「**クエリ オプティマイザー**」が、最適ではない遅い実行プランを選択してしまうことがあります。これは、SQL Server に限った話ではなく、Oracle や DB2 でも同様で、複雑なクエリになった場合は、クエリ オプティマイザーの動作に限界があるためです。

弊社の案件では、クエリの長さが 20KB（2 万文字）以上にもなるサブクエリで、印刷すると 10 ページにもなるようなクエリを扱ったことがあります。このクエリは、オプティマイザーが実行プランを選択するフェーズ（初回のコンパイル フェーズ）だけで、20 秒以上もの時間がかかり、かつ選択された実行プランも最適なものではない、遅い実行プランでした。このクエリに対しては、一時テーブルを利用してシンプルに記述し、その一時テーブルヘインデックスを作成するなどして、パフォーマンス向上を実現しました（同じ結果を何度か再利用する場合には、一時テーブルヘインデックスを作成することでパフォーマンスを向上させることができます）。このようなクエリは、メンテナンス性も非常に低く、実際のクエリ作成者以外が見たときに、誰も理解できないクエリ（誰も改修できないクエリ）となってしまうので、こういった状況にならないよう、一時テーブルなどを利用して、シンプルなクエリを記述することをお勧めします。

Note： 一時テーブルの照合順序、包含データベース（Contained Database）

一時テーブルの照合順序は、**SELECT INTO** で作成した場合は、もとのテーブルの照合順序を継承し、**CREATE TABLE** で作成した場合には **tempdb** データベースの照合順序が継承されます。

SQL Server 2012 からは、tempdb の照合順序に依存しない一時テーブルの作成ができる「**包含データベース**」（**Contained Database**）機能が提供されました。包含データベースを利用すれば、**CREATE TABLE** で作成した一時テーブルでも、データベースの照合順序を継承することができます。開発環境と本番環境で tempdb の照合順序が異なったりする場合でも、問題なく動作させることができますようになります。

2.10 テーブル変数

✦ テーブル変数とは

テーブル変数は、ローカル変数と同様、バッチ内でのみ有効なテーブルを作成できる機能です（バッチについては本自習書シリーズの「**Transact-SQL 入門**」編を参考にしてください）。テーブル変数は、次のようにデータ型へ「**table**」を指定し、CREATE TABLE ステートメントでの列定義と同様に、テーブルの定義が行えます。

```
DECLARE @テーブル変数名 table
( 列名1 データ型
, 列名2 データ型
, ...)
```

このように作成したテーブル変数は、明示的に削除する方法はなく、バッチの終了時に自動的に削除されます。

テーブル変数へ値を代入する場合には、次のように INSERT ステートメントのサブクエリを利用します。

```
INSERT INTO @テーブル変数名
SELECT * FROM テーブル名 ～～
```

✦ Let's Try

それでは、これを試してみましょう。

1. 前の Step と同様のクエリを、テーブル変数を利用するように置き替えてみましょう。

```
USE sampleDB
DECLARE @t table
( rownum int
, empno int
, empname char(50)
, sal int
, hiredate datetime
, deptno int )

INSERT INTO @t
SELECT ROW_NUMBER() OVER (ORDER BY hiredate DESC) AS rownum
, * FROM emp

SELECT * FROM @t
WHERE rownum BETWEEN 1 AND 3
```

```

DECLARE @t table
(
    rownum      int
    , empno     int
    , empname   char(50)
    , sal       int
    , hiredate  datetime
    , deptno    int )

INSERT INTO @t
SELECT ROW_NUMBER() OVER (ORDER BY hiredate DESC) AS rownum
,* FROM emp

SELECT * FROM @t
WHERE rownum BETWEEN 1 AND 3

```

| | rownum | empno | empname | sal | hiredate | deptno |
|---|--------|-------|-------------|--------|-------------------------|--------|
| 1 | 1 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 2 | 2 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |
| 3 | 3 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |

一時テーブルを利用した場合と同じ結果（入社日の新しい社員の 1 件目から 3 件目）を取得できていることを確認できます。

Note： テーブル変数と一時テーブルの使い分け

テーブル変数は、一時テーブルと比べて、次の制限事項があります。

- SELECT INTO でテーブル変数を作成できない
- テーブル変数へインデックスを作成できない
- パラレル処理の対象とならない（複数 CPU コアがある場合の並列処理）
- 有効範囲がバッチ内のみ（一時テーブルは接続している間有効）

このようにテーブル変数は、一時テーブルと比べて、利用が面倒なのと、パフォーマンス関連（インデックスが作成できない点とパラレル処理の対象とならない点）で一時テーブルよりも劣ります。単純な処理であれば、速度はほとんど変わりませんが、クエリ結果に対してインデックスを作成してチューニングしたい場合には、テーブル変数を利用することができません。したがって、テーブル変数を一時テーブルの置き換えとして利用しようと考えている場合は、置き換えることはせず、一時テーブルを利用することをお勧めします。

もちろん、単純な処理を記述する場合には、テーブル変数は便利ですし、次の STEP で説明するユーザー定義テーブル型として利用する場合には、大変便利な機能です。

Note： テーブル変数を配列のように利用する

Transact-SQL では、配列を扱える機能がないのですが、table データ型を利用すると、複数の値を格納できるので、配列と同じように利用することができます。これについては、次の STEP「ストアード プロシージャ」の入力パラメーターのところで説明します。

2.11 ユーザー定義テーブル型

✦ ユーザー定義テーブル型

ユーザー定義テーブル型 (User-Defined Table Type) は、table データ型のテーブル定義に対して、名前を付けてデータ型 (Type) として保存できる、SQL Server 2008 から提供された機能です。これは次のように利用します。

```
-- ユーザー定義テーブル型
CREATE TYPE 型名
AS TABLE
( 列名1 データ型
, 列名2 データ型
, ...)
go

-- ユーザー定義テーブル型の利用
DECLARE @変数名 型名
```

CREATE TYPE ステートメントで table データ型のテーブル定義に対して (ユーザー定義のデータ型として) 名前を付け、それを **DECLARE** でローカル変数を利用する際に利用できるようになります。**CREATE TYPE** と **DECLARE** は別々のバッチで (go で区切って) 利用する必要があります。作成したユーザー定義テーブル型は、データベース内へ永続化されるので、削除したい場合は、**DROP TYPE** ステートメントを利用して削除します。

✦ Let's Try

それでは、これを試してみましょう。

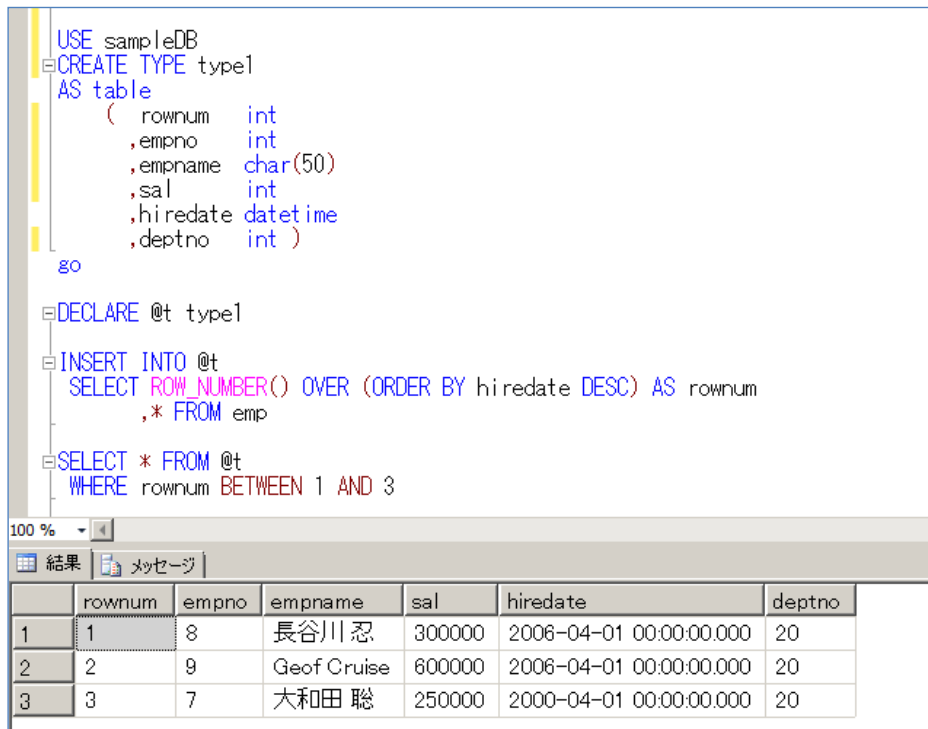
1. 前の STEP で利用したテーブル変数に対して、ユーザー定義テーブル型として保存してみましょう。

```
USE sampleDB
CREATE TYPE type1
AS table
( rownum int
, empno int
, empname char(50)
, sal int
, hiredate datetime
, deptno int )
go

DECLARE @t type1

INSERT INTO @t
SELECT ROW_NUMBER() OVER (ORDER BY hiredate DESC) AS rownum
, * FROM emp
```

```
SELECT * FROM @t
WHERE rownum BETWEEN 1 AND 3
```



```
USE sampleDB
CREATE TYPE type1
AS table
(
    rownum      int
    ,empno      int
    ,empname    char(50)
    ,sal        int
    ,hiredate   datetime
    ,deptno     int )
go

DECLARE @t type1

INSERT INTO @t
SELECT ROW_NUMBER() OVER (ORDER BY hiredate DESC) AS rownum
,* FROM emp

SELECT * FROM @t
WHERE rownum BETWEEN 1 AND 3
```

| | rownum | empno | empname | sal | hiredate | deptno |
|---|--------|-------|-------------|--------|-------------------------|--------|
| 1 | 1 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 2 | 2 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |
| 3 | 3 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |

前の STEP と同じ結果を取得できることを確認できます。このようにユーザー定義テーブル型（CREATE TYPE ステートメント）を利用すると、何度も利用するようなテーブル定義を簡単に再利用できるようになるので便利です。

- 作成したユーザー定義テーブル型を削除したい場合は、次のように DROP TYPE ステートメントを利用します。

```
DROP TYPE type1
```

ユーザー定義テーブル型は、次の STEP「ストアド プロシージャ」で説明する入力パラメーターのデータ型として利用することもできます。これについては、そのときに説明します。

2.12 CTE（共通テーブル式）

✦ CTE（Common Table Expression : 共通テーブル式）

CTE（共通テーブル式）は、一時テーブルやテーブル変数と似ていて、SELECT ステートメントで取得した結果に対して名前を付けることができる機能です。CTE は、SQL99 規格（1999 年に規格化された SQL 標準）に準拠した機能で、SQL Server 2005 からサポートされました。

CTE は、次のように利用します。

```
WITH 式名 [ (列名1, 列名2, ...) ]
AS
( SELECT ステートメント )
```

WITH に続けて名前を指定し、AS 以下へ SELECT ステートメントを記述します。また、WITH は、バッチの先頭で利用する必要があります。

✦ Let's Try

それでは、これを試してみましょう。

1. 一時テーブルとテーブル変数のところで利用したクエリを、CTE を利用するように置き換えてみましょう。

```
USE sampleDB
go
WITH cteTest1
AS
(
    SELECT ROW_NUMBER() OVER (ORDER BY hiredate DESC) AS rownum
    , * FROM emp
)

SELECT * FROM cteTest1
WHERE rownum BETWEEN 1 AND 3
```

WITH cteTest1

```
AS
(
    SELECT ROW_NUMBER() OVER (ORDER BY hiredate DESC) AS rownum
    , * FROM emp
)

SELECT * FROM cteTest1
WHERE rownum BETWEEN 1 AND 3
```

100 %

結果 メッセージ

| | rownum | empno | empname | sal | hiredate | deptno |
|---|--------|-------|-------------|--------|-------------------------|--------|
| 1 | 1 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 2 | 2 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |
| 3 | 3 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |

一時テーブルやテーブル変数、インライン ビューを利用した場合（STEP 2.7～2.11）と同じ結果（入社日の新しい社員の 1 件目から 3 件目）を取得できていることを確認できます。

Note：CTE と一時テーブル、テーブル変数、インライン ビューとの使い分け

CTE は、一時テーブルやテーブル変数と同じように SELECT ステートメントの結果に対して、名前を付けて保存できる機能です。内部的な動作は、インライン ビューとほぼ同じなので、単純なクエリの場合は、一時テーブルよりもパフォーマンスが良く処理される場合が多くあります。しかし、インライン ビューのときと同様、複雑なクエリになった場合には、逆の結果になることが多々ありますので、気を付けてください。前述したように、サブクエリの入れ子が何段階にもなっている複雑なクエリで、かつ結果を何度も利用するような場合には、（インデックスを付与した）一時テーブルを利用することをお勧めします。

したがって、CTE は、インライン ビューの置き換え（インライン ビューを読みやすくするためなど）として利用したい場合にお勧めの機能です。

なお、CTE では、CTE にしかない特徴として、次の STEP 2.13 で説明する「再帰クエリ」という利用方法が可能です。再帰クエリを利用する場合には、CTE は大変便利です。ぜひ活用してみてください。

2.13 再帰クエリ (CTE)

再帰クエリ

再帰クエリは、その名のとおり、SELECT ステートメントで取得した結果セットに対して、再帰的に、繰り返し呼び出すクエリのことを指します。CTE（共通テーブル式）を利用すると、この再帰クエリを実現することができます。このようなクエリは、親子階層をもったテーブルの場合に役立ちます。

Let's Try

それでは、これを試してみましょう。

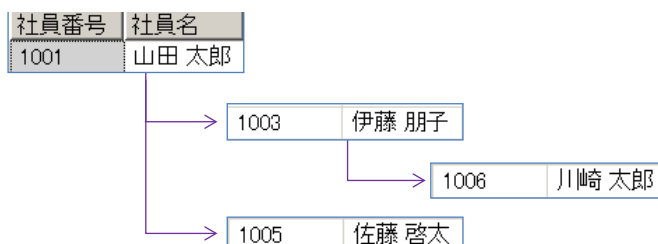
- まずは、次のような親子階層をもった社員テーブル（「**上司社員番号**」列に上司の社員番号を格納）を作成します。CREATE TABLE や INSERT ステートメントなどは、サンプル スクリプト ファイル内の「**Step2_Script.sql**」へ記述してあるので、そこからコピーして実行することができます。

| 社員番号 | 社員名 | 上司社員番号 | 性別 |
|------|-------|--------|----|
| 1001 | 山田 太郎 | NULL | 男性 |
| 1002 | 鈴木 一郎 | NULL | 男性 |
| 1003 | 伊藤 朋子 | 1001 | 女性 |
| 1004 | 若旅 素子 | 1002 | 女性 |
| 1005 | 佐藤 啓太 | 1001 | 男性 |
| 1006 | 川崎 太郎 | 1003 | 男性 |

```
CREATE TABLE 社員
( 社員番号      int      NOT NULL PRIMARY KEY,
  社員名        varchar(40) NULL,
  上司社員番号  int      NULL,
  性別          char(4)  NULL )

INSERT INTO 社員 VALUES (1001, '山田 太郎', NULL, '男性')
INSERT INTO 社員 VALUES (1002, '鈴木 一郎', NULL, '男性')
INSERT INTO 社員 VALUES (1003, '伊藤 朋子', 1001, '女性')
INSERT INTO 社員 VALUES (1004, '若旅 素子', 1002, '女性')
INSERT INTO 社員 VALUES (1005, '佐藤 啓太', 1001, '男性')
INSERT INTO 社員 VALUES (1006, '川崎 太郎', 1003, '男性')
```

この社員データには、次の階層（上司と部下）があります。

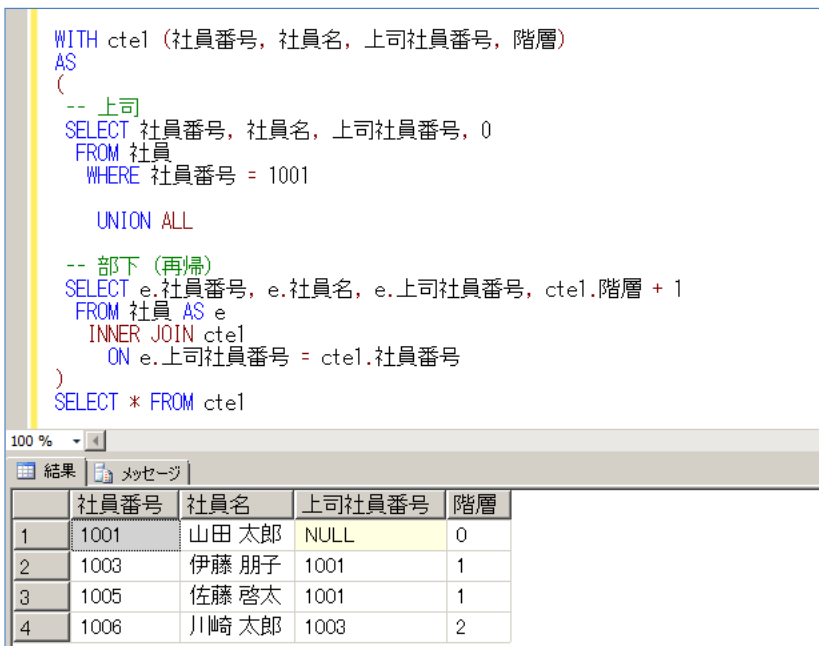


2. 次に、CTE（共通テーブル式）を利用して、再帰クエリを実行し、この階層（階層のレベル）を取り出してみましょう。

```
WITH cte1 (社員番号, 社員名, 上司社員番号, 階層)
AS
(
  -- 上司
  SELECT 社員番号, 社員名, 上司社員番号, 0
  FROM 社員
  WHERE 社員番号 = 1001

  UNION ALL

  -- 部下（再帰）
  SELECT e.社員番号, e.社員名, e.上司社員番号, cte1.階層 + 1
  FROM 社員 AS e
  INNER JOIN cte1
  ON e.上司社員番号 = cte1.社員番号
)
SELECT * FROM cte1
```



```
WITH cte1 (社員番号, 社員名, 上司社員番号, 階層)
AS
(
  -- 上司
  SELECT 社員番号, 社員名, 上司社員番号, 0
  FROM 社員
  WHERE 社員番号 = 1001

  UNION ALL

  -- 部下（再帰）
  SELECT e.社員番号, e.社員名, e.上司社員番号, cte1.階層 + 1
  FROM 社員 AS e
  INNER JOIN cte1
  ON e.上司社員番号 = cte1.社員番号
)
SELECT * FROM cte1
```

| | 社員番号 | 社員名 | 上司社員番号 | 階層 |
|---|------|-------|--------|----|
| 1 | 1001 | 山田 太郎 | NULL | 0 |
| 2 | 1003 | 伊藤 朋子 | 1001 | 1 |
| 3 | 1005 | 佐藤 啓太 | 1001 | 1 |
| 4 | 1006 | 川崎 太郎 | 1003 | 2 |

UNION ALL で上司と部下の結果を統合し、結合条件（部下側の INNER JOIN の ON 句で指定する結合条件）で上司の社員番号と再帰クエリ（CTE で繰り返し取得している社員番号）を指定することで、親子階層のレベルを取得できるようになっています。

このように、CTE を利用すると、親子階層を簡単に取得できるようになるので便利です。なお、次に説明する **HierarchyID** データ型を再帰クエリと組み合わせて利用すると、親子階層のパスまで取得することが可能です。

2.14 HierarchyID データ型

✦ HierarchyID データ型

HierarchyID は、階層（Hierarchy）のパスを取得／操作が可能なデータ型で、SQL Server 2008 から提供された機能です。このデータ型には、**GetRoot** メソッドや、**Path** プロパティが用意されていて、親子階層のパスが操作できるようになっています。

✦ Let's Try

それでは、これを試してみましょう。

1. 前の Step で利用した「社員」テーブルに対して、次のように HierarchyID データ型を利用して、再帰クエリを実行してみましょう。

```
WITH cte1 (path, 社員番号, 社員名, 上司社員番号, 階層)
AS
(
  -- 上司
  SELECT  HierarchyID::GetRoot() AS root
        , 社員番号, 社員名, 上司社員番号, 0
  FROM 社員
  WHERE 社員番号 = 1001

  UNION ALL

  -- 部下（再帰）
  SELECT  CAST( cte1.path.ToString()
              + CAST(e.社員番号 AS varchar(30))
              + '/' AS HierarchyID )
        , e.社員番号, e.社員名, e.上司社員番号, cte1.階層+ 1
  FROM 社員 AS e
  INNER JOIN cte1
  ON e.上司社員番号 = cte1.社員番号
)
SELECT path.ToString(), * FROM cte1
```

| (列名なし) | path | 社員番号 | 社員名 | 上司社員番号 | 階層 |
|-------------|--------------|------|-------|--------|----|
| / | 0x | 1001 | 山田 太郎 | NULL | 0 |
| /1003/ | 0xEE2DC0 | 1003 | 伊藤 朋子 | 1001 | 1 |
| /1005/ | 0xEE2EC0 | 1005 | 佐藤 啓太 | 1001 | 1 |
| /1003/1006/ | 0xEE2DFB8BD0 | 1006 | 川崎 太郎 | 1003 | 2 |

HierarchyID により
階層のパスを取得

このように HierarchyID データ型を利用すると、親子階層のパスを簡単に取得できるので便利です。

STEP 3. ストアド プロシージャ

この STEP では、ストアド プロシージャについて説明します。ストアド プロシージャの基本から、「入力パラメーター」や「出力パラメーター」、「RETURN コード」、「テーブル値パラメーター」、「IDENTITY プロパティの注意点」などを説明します。

この STEP では、次のことを学習します。

- ✓ ストアド プロシージャとは
- ✓ ストアド プロシージャの作成と実行
- ✓ 入力パラメーター
- ✓ テーブル値パラメーター
- ✓ 出力パラメーター
- ✓ IDENTITY プロパティの注意点
- ✓ RETURN コード

3.1 ストアド プロシージャとは

◆ ストアド プロシージャとは

ストアド プロシージャ (Stored Procedure) は、まとめて処理したいデータベース操作を 1 つのオブジェクトとして SQL Server 上に保存したものです。Store は「保存する」、「蓄える」、Procedure は「手続き」、「手順」という意味です。

ストアド プロシージャを利用するメリットは、次の 3 つです。

1. テーブル構造の隠蔽

ストアド プロシージャを利用すると、ストアド プロシージャ経由でのみテーブル操作を行えるようにすることができます。これにより、テーブルに対する直接の操作を拒否（テーブルに対する操作権限を REVOKE または DENY）することができるので、一般ユーザーからテーブルを直接操作されることを防ぐことができます。

2. アプリケーション ロジックの共有化

ストアド プロシージャは、任意のアプリケーションから呼び出すことができるので、同じような処理を行うアプリケーションを複数作成する場合には、その部分を共有化できます。

3. パフォーマンスの向上

ストアド プロシージャを利用すると、コンパイル済みの実行プラン（クエリ オプティマイザーが選択した最適な実行プラン）をプロシージャ キャッシュへ格納できるようになるので、SQL ステートメントを解釈（コンパイル）するオーバーヘッドを軽減できます。

また、ストアド プロシージャは、ネットワーク上を流れる SQL ステートメントを少なくすることもできます。

3.2 ストアド プロシージャの作成と実行

✦ ストアド プロシージャの作成 ～CREATE PROCEDURE～

ストアド プロシージャは、**CREATE PROCEDURE** ステートメントを利用して作成します。構文は、次のとおりです。

```
CREATE PROCEDURE ストアドプロシージャ名
AS
任意の Transact-SQL ステートメント
```

CREATE PROCEDURE に続けてストアド プロシージャの名前を記述し、**AS** 以下に任意の Transact-SQL ステートメントを記述します。**PROCEDURE** は、**PROC** と省略することもできます。また、CREATE PROCEDURE ステートメントは、バッチの先頭で記述する必要があります。

✦ ストアド プロシージャの実行 ～EXECUTE～

作成したストアド プロシージャを実行するには、**EXECUTE** ステートメントを利用して、次のように記述します。

```
EXECUTE ストアドプロシージャ名
```

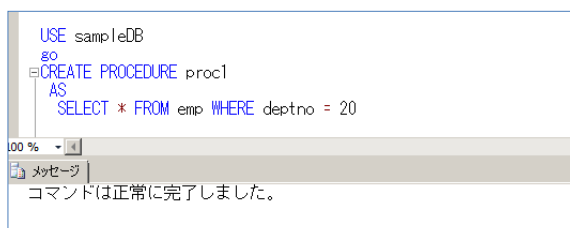
EXECUTE は、動的 SQL のときに利用したのと同じステートメントで、「**EXEC**」と省略することも可能です。なお、バッチの先頭の場合は、EXEC を付けずに、ストアド プロシージャの名前だけで実行することもできます。

✦ Let's Try

それでは、ストアド プロシージャを作成して、実行してみましょう。

1. まずは、「**sampleDB**」データベースの「**emp**」テーブルから「**deptno**」（部門番号）が **20** の社員を取得するストアド プロシージャ（名前は **proc1**）を作成します。

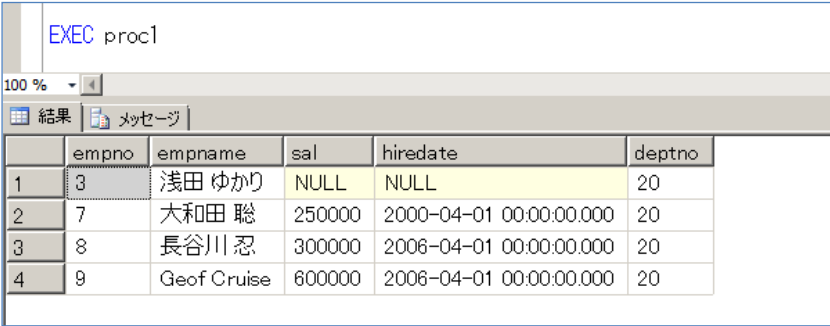
```
USE sampleDB
go
CREATE PROCEDURE proc1
AS
SELECT * FROM emp WHERE deptno = 20
```



USE sampleDB の後の「go」を忘れずに実行するようにしてください（CREATE PROCEDURE は、バッチの先頭に記述する必要があります）。ストアード プロシージャの名前は、「**proc1**」としています。

2. 次に、作成したストアード プロシージャを実行してみましょう。

```
EXEC proc1
```



EXEC proc1

100 %

結果 メッセージ

| | empno | empname | sal | hiredate | deptno |
|---|-------|-------------|--------|-------------------------|--------|
| 1 | 3 | 浅田 ゆかり | NULL | NULL | 20 |
| 2 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |
| 3 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 4 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |

emp テーブルから **deptno**（部門番号）が **20** の社員を取得できたことを確認できます。

3.3 入力パラメーター

✦ 入力パラメーター

ストアド プロシージャは、入力パラメーターを利用すると、汎用的なストアド プロシージャを作成できるようになります。入力パラメーターは、次のように利用します。

```
CREATE PROCEDURE ストアドプロシージャ名
    @パラメーター名1 データ型 [ = 初期値]
    , @パラメーター名2 データ型 [ = 初期値], ...
AS
    任意の Transact-SQL ステートメント
```

パラメーターの名前は、ローカル変数の場合と同じように先頭に「@」を付けて、データ型を指定します。データ型の隣に「=」を記述した場合は、初期値を設定することもできます。

パラメーター付きのストアド プロシージャを実行する場合は、次のように記述します。

```
EXEC ストアドプロシージャ名 @パラメーター名1 = 値1, @パラメーター名2 = 値2, ...
または
EXEC ストアドプロシージャ名 値1, 値2, ...
```

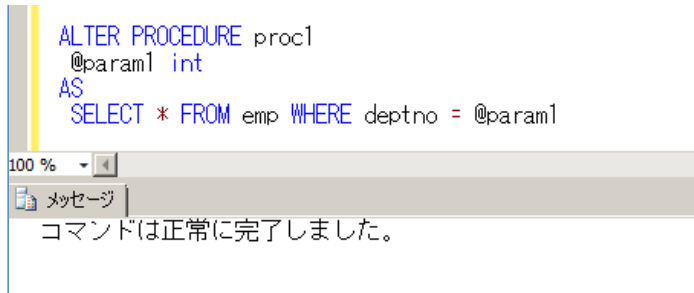
パラメーター名を記述して、「=」に続けて代入したい値を指定する方法と、ストアド プロシージャ内で定義されたパラメーターの順番に、左から値をカンマ区切りで指定する方法の 2 つがあります。

✦ Let's Try

それでは、入力パラメーターを試してみましょう。

1. 前の Step で作成した「**proc1**」ストアド プロシージャは、「**deptno**」（部門番号）が **20** の社員のみを取得するストアド プロシージャで、部門番号が“固定”でしたので、これを入力パラメーターを利用して、汎用的なストアド プロシージャに変更してみましょう。既存のストアド プロシージャを変更するには、「**ALTER PROCEDURE**」ステートメントを利用します。

```
ALTER PROCEDURE proc1
    @param1 int
AS
    SELECT * FROM emp WHERE deptno = @param1
```

パラメーター名を「@param1」、データ型を「int」として、これを deptno の検索条件でパラメーター化しています。

2. 次に、このストアード プロシージャを実行してみましょう。

```
EXEC proc1 @param1=20
または
EXEC proc1 20
```

EXEC proc1 @param1=20

100 %

結果 メッセージ

| | empno | empname | sal | hiredate | deptno |
|---|-------|-------------|--------|-------------------------|--------|
| 1 | 3 | 浅田 ゆかり | NULL | NULL | 20 |
| 2 | 7 | 大和田 聡 | 250000 | 2000-04-01 00:00:00.000 | 20 |
| 3 | 8 | 長谷川 忍 | 300000 | 2006-04-01 00:00:00.000 | 20 |
| 4 | 9 | Geof Cruise | 600000 | 2006-04-01 00:00:00.000 | 20 |

deptno（部門番号）が **20** の社員のみを取得できていることを確認できます。

3. 次に、パラメーターに与える値を **10** へ変更して実行してみましょう。

```
EXEC proc1 @param1=10
または
EXEC proc1 10
```

EXEC proc1 10

100 %

結果 メッセージ

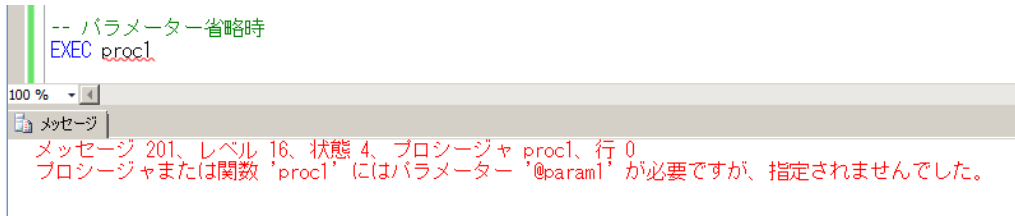
| | empno | empname | sal | hiredate | deptno |
|---|-------|---------|--------|-------------------------|--------|
| 1 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |
| 2 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 3 | 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |
| 4 | 5 | 中野 浩之 | 500000 | 1996-04-01 00:00:00.000 | 10 |
| 5 | 6 | 内藤 太郎 | 500000 | 1997-04-01 00:00:00.000 | 10 |

今度は、部門番号が **10** の社員のみを取得できていることを確認できます。

このように、入力パラメーターを利用すると、ストアード プロシージャの実行時に値を指定できるようになるので、汎用的なストアード プロシージャを作成することができます。

✦ パラメーター省略時のエラーと初期値の設定

入力パラメーターを利用している場合、パラメーターを省略して実行しようとする、次のエラーが発生します。



このエラーを回避するには、入力パラメーターへ初期値を設定する必要があります。では、これを試してみましょう。

1. ALTER TABLE ステートメントを利用して、「**proc1**」ストアード プロシージャの入力パラメーター「**@param1**」の初期値を「**10**」へ設定してみましょう。

```
ALTER PROCEDURE proc1
    @param1 int = 10
AS
SELECT * FROM emp WHERE deptno = @param1
```

2. 変更後、入力パラメーターを省略して実行してみましょう。

```
EXEC proc1
```

| | empno | empname | sal | hiredate | deptno |
|---|-------|---------|--------|-------------------------|--------|
| 1 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |
| 2 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 3 | 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |
| 4 | 5 | 中野 浩之 | 500000 | 1996-04-01 00:00:00.000 | 10 |
| 5 | 6 | 内藤 太郎 | 500000 | 1997-04-01 00:00:00.000 | 10 |

初期値「10」が補われて、deptno（部門番号）が 10 の社員のみを取得できたことを確認できます。

✦ パラメーターの入力チェック

パラメーターは、入力チェックを行うことも可能です。これは、パラメーターの初期値を“**NULL**”へ設定して、NULL かどうかをチェックする IF 分岐を追加するだけで簡単に実現できます。

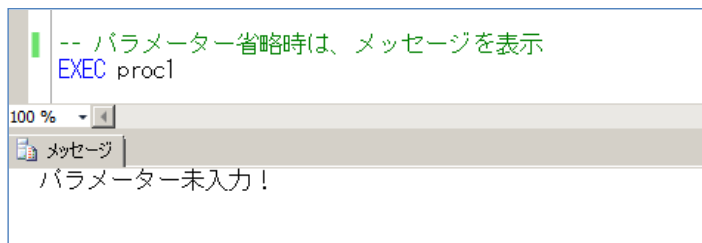
それでは、これを試してみましょう。

1. 次のように ALTER PROCEDURE ステートメントを利用して「proc1」ストアード プロシージャを変更します。

```
ALTER PROCEDURE proc1
    @param1 int = NULL
AS
    IF @param1 IS NULL
        BEGIN
            PRINT 'パラメーター未入力！'
        END
    ELSE
        BEGIN
            SELECT * FROM emp WHERE deptno = @param1
        END
END
```

2. 変更後、入力パラメーターを省略して実行してみましょう。

```
EXEC proc1
```



PRINT ステートメントで指定したメッセージが表示されたことを確認できます。このようにパラメーターの初期値を NULL にし、それかどうかを判断するようにすれば、パラメーターの入力チェックとして利用できるようになります。

Note : RAISERROR によるエラーの発生

PRINT ステートメントで出力したメッセージは、VB や C# などのアプリケーションから取得するには少々面倒です。これを回避するには、「**RAISERROR**」というステートメントを使用してユーザー定義のエラーを発生させることです。この場合は、エラーとしてアプリケーションへ通達されるので、アプリケーション側のハンドリングも簡単に行うことができます。RAISERROR ステートメントについては、次の STEP で説明します。

➡ RETURN による強制終了

1 つ前の例で試したパラメーターの入力チェックでは、入力チェックを通過した場合の処理を ELSE 以下の BEGIN と END で囲まなければならない、処理内容が多い場合には、分かりづらくなります。これを分かりやすくするには、「**RETURN**」ステートメントを利用します。RETURN は、ストアード プロシージャを強制終了することができるステートメントです。

それでは、これを試してみましょう。

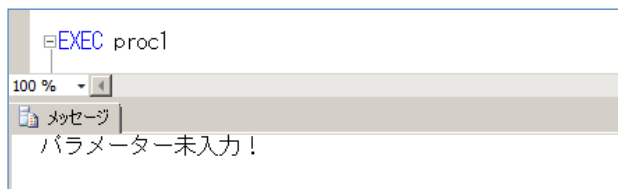
1. 次のように ALTER PROCEDURE ステートメントを利用して「proc1」ストアード プロシージャ

ヤを変更します。

```
ALTER PROCEDURE proc1
    @param1 int = NULL
AS
    IF @param1 IS NULL
    BEGIN
        PRINT 'パラメーター未入力！'
        RETURN
    END
    SELECT * FROM emp WHERE deptno = @param1
```

2. 変更後、入力パラメーターを省略して実行してみましょう。

```
EXEC proc1
```



このように RETURN ステートメントを追加すると、パラメーターの入力チェックを通過しなかった場合にストアド プロシージャを強制終了できるようになるので、通過した場合の処理を ELSE 以下へ記述しなくて済むようになります。

Note : RETURN ステートメントはリターン コードを指定可能

詳しくは、次の Step で説明しますが、RETURN ステートメントでは、**RETURN(0)** や **RETURN(1)** のように記述して、リターン コードを指定することもできます。

3.4 IN 演算子のパラメーター化

✦ IN 演算子のパラメーター化

WHERE 句の条件式に利用する IN 演算子をパラメーター化する方法は、簡単そうであるが、実は簡単ではありません。

✦ Let's Try

それでは、これを試してみましょう。

1. まずは、次のように IN 演算子を利用して「empno」（社員番号）を検索する SELECT ステートメントをストアド プロシージャ化してみます。

```
USE sampleDB
go
CREATE PROCEDURE proc2
    @param1 int
AS
    SELECT * FROM emp WHERE empno IN (@param1)
```

2. 次に、@param1 へ「1」を指定して、「proc2」ストアド プロシージャを実行してみよう。

```
proc2 1
```

| | empno | empname | sal | hiredate | deptno |
|---|-------|---------|--------|-------------------------|--------|
| 1 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |

3. 続いて、@param1 へ「1, 5」を指定して、「proc2」ストアド プロシージャを実行してみよう。

```
proc2 1, 5
```

| | empno | empname | sal | hiredate | deptno |
|---|-------|---------|--------|-------------------------|--------|
| 1 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |

結果は、エラーになり、ストアド プロシージャの実行が失敗します。ストアド プロシージャでは、パラメーターの指定時に「,」を利用すると、パラメーターの区切りとみなされるから

です。したがって、IN 演算子をパラメーター化する場合には、値の分だけパラメーターを用意するか、Step1 で説明した「動的 SQL」を利用して文字列として SQL を組み立てるか、後述の「テーブル値パラメーター」という機能を利用しなければなりません。

Note : IN 演算子へ与える値の分だけパラメーターを用意する方法

IN 演算子へ与える値の分だけパラメーターを用意する場合は、次のように作成します。

```

CREATE PROCEDURE procX
    @param1 int, @param2 int
AS
    SELECT * FROM emp WHERE empno IN (@param1, @param2)
go

procX 1, 5

```

| | empno | empname | sal | hiredate | deptno |
|---|-------|---------|--------|-------------------------|--------|
| 1 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |
| 2 | 5 | 中野 浩之 | 500000 | 1996-04-01 00:00:00.000 | 10 |

しかし、この方法では、値の数が増えた場合には、パラメーターの数が増えますし、値の数が可変の場合には対応できません。したがって、この方法は、お勧めではありません。次の Step3.5 で説明するテーブル値パラメーターを利用することをお勧めします。

Note : 動的 SQL を利用する場合

動的 SQL を利用して IN 演算子をパラメーター化する場合は、次のように記述します。

```

CREATE PROCEDURE procY
    @param1 varchar(100)
AS
    EXEC ('SELECT * FROM emp WHERE empno IN (' + @param1 + ')')
go

procY @param1 = '1, 5'

```

| | empno | empname | sal | hiredate | deptno |
|---|-------|---------|--------|-------------------------|--------|
| 1 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |
| 2 | 5 | 中野 浩之 | 500000 | 1996-04-01 00:00:00.000 | 10 |

パラメーターを **varchar** 型で定義し、文字列として IN 演算子へ与える値を指定すれば、動的 SQL として複数の値を指定することができます。しかし、この方法では、文字列として SQL を組み立てるため、実行する SQL が長い場合には、メンテナンス性が悪く、読みづらいコードとなってしまいます。したがって、この方法も、お勧めではありません。

3.5 テーブル値パラメーターとユーザー定義テーブル型

✦ テーブル値パラメーターとユーザー定義テーブル型

「**テーブル値パラメーター**」(Table-Valued Parameters) は、Step 2.11 で説明した「**ユーザー定義テーブル型**」をストアード プロシージャの入力パラメーターとして指定できる機能で、SQL Server 2008 から提供された機能です。これを利用すると、IN 演算子をパラメーター化する場合に非常に便利です。

テーブル値パラメーターは、次のように利用します。

```
-- テーブル値パラメーター (ユーザー定義テーブル型を入力パラメーターとして利用)
CREATE PROCEDURE ストアド プロシージャ名
    @パラメーター名 ユーザー定義テーブル型 READONLY
, ...
AS
    任意の Transact-SQL ステートメント
```

パラメーターのデータ型を指定するところへユーザー定義テーブル型を指定し、**READONLY** (読み取り専用) キーワードを付与して利用します。

ユーザー定義テーブル型は、Step 2.11 で説明したように table データ型のテーブル定義に対して、名前を付けてデータ型 (Type) として保存できる機能です。復習になりますが、ユーザー定義テーブル型は、次のように利用します。

```
-- ユーザー定義テーブル型
CREATE TYPE 型名
AS TABLE
( 列名1 データ型
, 列名2 データ型
, ... )
go

-- ユーザー定義テーブル型の利用
DECLARE @変数名 型名
```

✦ Let's Try

それでは、テーブル値パラメーターを利用して、IN 演算子をパラメーター化してみましょう。

1. まずは、IN 演算子へ与える値を格納するための **int** データ型の列を持ったユーザー定義テーブル型を「**valuelist**」という名前で作成します。

```
-- ユーザー定義テーブル型
CREATE TYPE valuelist
AS TABLE ( val int )
go
```

2. 次にテーブル値パラメーターを利用して、ストアード プロシージャを作成します。

```
-- テーブル値パラメーターを利用するストアード プロシージャ
CREATE PROCEDURE proc3
    @v valuelist READONLY
AS
    SELECT * FROM emp
    WHERE empno IN ( SELECT val FROM @v )
```

IN 演算子の部分をサブクエリとし、@v（ユーザー定義テーブル型 valuelist を指定したパラメーター）から val 列（int データ型の列）を取得して、それを IN 演算子へ与えています。

3. 次に、ユーザー定義テーブル型の val 列へ「1」と「5」を格納して、この値をストアード プロシージャのパラメーターへ与えてみます。

```
-- ユーザー定義テーブル型を利用した変数@v の宣言
DECLARE @v AS valuelist

-- 変数 @v へ値の格納（IN 演算子に与える値）
INSERT INTO @v (val)
VALUES ( 1 )
        , ( 5 )

-- ストアド プロシージャの実行
EXEC proc3 @v
```

```
-- ユーザー定義テーブル型を利用した変数 @v の宣言
DECLARE @v AS valuelist

-- 変数 @v へ値の格納（IN 演算子に与える値）
INSERT INTO @v (val)
VALUES ( 1 )
        , ( 5 )

-- ストアド プロシージャの実行
EXEC proc3 @v
```

100 %

結果 メッセージ

| | empno | empname | sal | hiredate | deptno |
|---|-------|---------|--------|-------------------------|--------|
| 1 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |
| 2 | 5 | 中野 浩之 | 500000 | 1996-04-01 00:00:00.000 | 10 |

empno が 1 と 5 の社員を取得できていることを確認できます。

このように、ユーザー定義テーブル型とテーブル値パラメーターを利用すると、IN 演算子のパラメーター化が簡単に行えるようになります。

Tips : ADO.NET からテーブル値パラメーターを利用する場合

VB や C# などのアプリケーションからテーブル値パラメーターへ値を利用する場合は、**SqlParameter** クラスのデータ型として「**Structured**」を指定して、**DataTable** クラスへ値を格納しておくようにします。具体的には次のように利用します（VB の場合）。

```
Imports System.Data
```



```
Imports System.Data.SqlClient
Imports System.Data.SqlTypes
:
' DataTable オブジェクトへ値を格納
Dim dt As New DataTable
Dim row As DataRow
dt.Columns.Add("val", Type.GetType("System.Int32"))

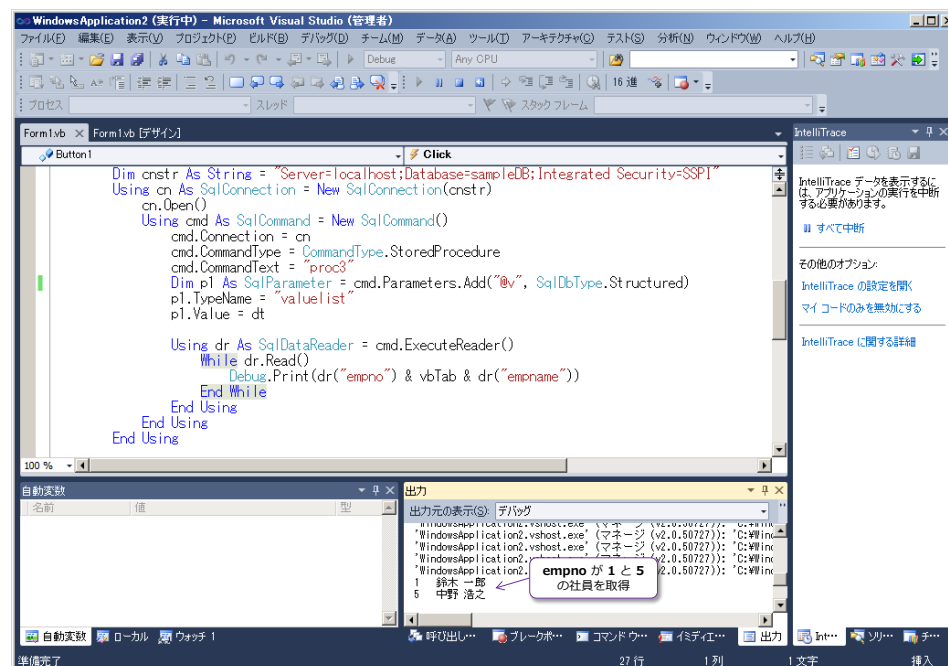
' val 列へ 1 を格納
row = dt.NewRow()
row("val") = 1
dt.Rows.Add(row)

' val 列へ 5 を格納
row = dt.NewRow()
row("val") = 5
dt.Rows.Add(row)

Dim cnstr As String = "Server=localhost;Database=sampleDB;Integrated Security=SSPI"
Using cn As SqlConnection = New SqlConnection(cnstr)
    cn.Open()
    Using cmd As SqlCommand = New SqlCommand()
        cmd.Connection = cn
        cmd.CommandType = CommandType.StoredProcedure
        cmd.CommandText = "proc3"

        ' SqlParameter のデータ型へ Structured を指定
        Dim p1 As SqlParameter = cmd.Parameters.Add("@v", SqlDbType.Structured)
        p1.TypeName = "valuelist"
        p1.Value = dt

        Using dr As SqlDataReader = cmd.ExecuteReader()
            While dr.Read()
                Debug.Print(dr("empno") & vbTab & dr("empname"))
            End While
        End Using
    End Using
End Using
```



3.6 出力パラメーター (OUTPUT)

✦ 出力パラメーター

ストアド プロシージャでは、「出力パラメーター」を利用して、値を返すこともできます。これは、次のように利用します。

```
CREATE PROCEDURE ストアドプロシージャ名
    @パラメーター名 データ型 OUTPUT
    ...
AS
    任意の Transact-SQL ステートメント
```

データ型の隣に OUTPUT キーワードを付けると出力パラメーターになります。

ストアド プロシージャの実行時に、出力パラメーターを取得するには、次のように利用します。

```
DECLARE @変数名 データ型
EXEC ストアドプロシージャ名 @パラメーター名 = 変数名 OUTPUT, ...
```

出力パラメーターを受け取るためのローカル変数を宣言し、ストアド プロシージャの実行時に出力パラメーターを OUTPUT キーワードを指定します。

✦ Let's Try

それでは、これを試してみましょう。

1. まずは、次のように「emp」テーブルを「deptno」（部門番号）で絞り込んで取得し、その結果件数を「@@ROWCOUNT」というシステム関数で取得し、それを出力パラメーター「@param2」として出力します。

```
USE sampleDB
go
CREATE PROCEDURE proc4
    @param1 int,
    @param2 int OUTPUT
AS
    SELECT * FROM emp WHERE deptno = @param1
    SELECT @param2 = @@ROWCOUNT
```

Note : @@ROWCOUNT で影響のあった行数の取得

@@ROWCOUNT は、1 つ前の SQL ステートメントを実行したときに影響のあった行数 (Row Count) を返すことができるシステム関数です。非常に便利な関数なので、覚えておくと役立ちます。

2. 次に、ストアド プロシージャを実行して、出力パラメーターを取得してみましょう。

```

DECLARE @out1 int
EXEC proc4 10, @out1 OUTPUT

SELECT @out1

```

-- 出力パラメーターを受け取るためのローカル変数

The screenshot shows a SQL query window with the following code:

```

DECLARE @out1 int
EXEC proc4 10, @out1 OUTPUT

SELECT @out1

```

Below the code, the 'Results' pane displays a table with 5 rows and 5 columns: empno, empname, sal, hiredate, and deptno. The data is as follows:

| | empno | empname | sal | hiredate | deptno |
|---|-------|---------|--------|-------------------------|--------|
| 1 | 1 | 鈴木 一郎 | 500000 | 1998-04-01 00:00:00.000 | 10 |
| 2 | 2 | 小田 良夫 | 300000 | 1999-04-01 00:00:00.000 | 10 |
| 3 | 4 | 田村 健一 | 700000 | 1985-04-01 00:00:00.000 | 10 |
| 4 | 5 | 中野 浩之 | 500000 | 1996-04-01 00:00:00.000 | 10 |
| 5 | 6 | 内藤 太郎 | 500000 | 1997-04-01 00:00:00.000 | 10 |

A red bracket on the right side of the table highlights the 'deptno' column, with a callout box stating: 'deptno が 10 の社員 5件のデータ'.

Below the table, the 'Messages' pane shows the output of the SELECT statement: '5'.

A callout box points to the '5' in the Messages pane, stating: '出力パラメーターで取得した @@ROWCOUNT の結果'.

このように出力パラメーターを利用すると、ストアード プロシージャから結果を返すことができます。

Tips : ADO.NET で出力パラメーターを取得する方法

ADO.NET (VB) から出力パラメーターを取得するには、次のように **SqlParameter** クラスで **Direction** プロパティを **ParameterDirection.Output** へ指定するようにします。また、**SqlDataReader** を利用して結果を取得している場合は、これを **Close (End Using)** してから **Value** プロパティを参照する必要があります。

```

Imports System.Data
Imports System.Data.SqlClient
:
Using cn As New SqlConnection("Server=localhost;Database=sampleDB;Integrated Security=SSPI;")
    Using cmd As New SqlCommand("proc4", cn)
        cmd.CommandType = CommandType.StoredProcedure

        ' 入力パラメーター (@param1)
        Dim p1 As SqlParameter = cmd.Parameters.Add("@param1", SqlDbType.Int)
        p1.Value = Me.TextBox1.Text

        ' 出力パラメーター (@param2) の Direction プロパティを設定
        Dim p2 As SqlParameter = cmd.Parameters.Add("@param2", SqlDbType.Int)
        p2.Direction = ParameterDirection.Output
        Try
            cn.Open()
            Using dr As SqlDataReader = cmd.ExecuteReader()
                While dr.Read()
                    Me.ListBox1.Items.Add( dr("empname").ToString() )
                End While
            End Using

            ' 出力パラメーターの取得 (SqlDataReader の End Using 後に記述することに注意)
            Me.Label2.Text = p2.Value & " 件のデータがありました"

        Catch ex As Exception
            ' MessageBox.Show(ex.Message)
        Finally
            cn.Close()
        End Try
    End Using
End Using

```

3.7 出力パラメーターで IDENTITY 値の取得

✦ 出力パラメーターで IDENTITY 値の取得

出力パラメーターは、IDENTITY プロパティの値を取得する場合によく利用します。IDENTITY プロパティは、自動採番を行える機能で、テーブルの作成時に、次のように利用します。

```
CREATE TABLE テーブル名
( 列名1 データ型 IDENTITY(初期値, 増分)
, 列名2 データ型 )
```

データ型に続けて IDENTITY と指定すると、その列へ自動的に番号を振ることができるようになります。IDENTITY プロパティは、「**IDENTITY(1,1)**」と指定すれば、1 から 1 ずつ増えていく番号 (1、2、3、…) を自動で振れるようになります。

自動採番された IDENTITY 値を取得するには「**SCOPE_IDENTITY**」というシステム関数を利用します。

Note : IDENTITY は、Oracle でのシーケンス、Access でのオートナンバー

IDENTITY プロパティは、Oracle での「シーケンス」(SEQUENCE、順序)、Access での「オートナンバー」に相当する機能です。なお、SQL Server 2012 から、Oracle と同じように利用できる「シーケンス」機能がサポートされるようになりました。

✦ Let's Try

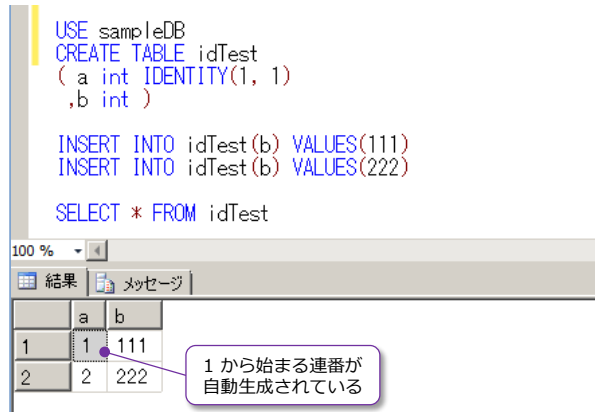
それでは、IDENTITY プロパティを試してみましょう。

1. まずは、「idTest」という名前のテーブルを作成し、「a」列に対して IDENTITY プロパティを「**IDENTITY(1,1)**」と指定し、データを 2 件 INSERT してみます。

```
USE sampleDB
CREATE TABLE idTest
( a int IDENTITY(1, 1)
, b int )

INSERT INTO idTest(b) VALUES(111)
INSERT INTO idTest(b) VALUES(222)

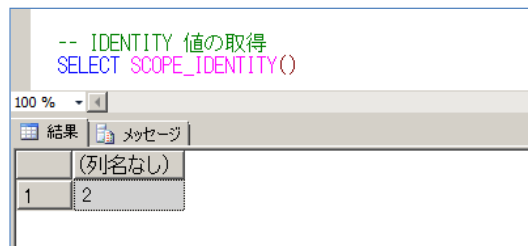
SELECT * FROM idTest
```



a 列には、1、2 と 1 から始まる連番が格納されたことを確認できます。

- 次に、自動採番された IDENTITY 値を **SCOPE_IDENTITY** 関数で取得してみましょう。

```
SELECT SCOPE_IDENTITY()
```



最後に自動生成された値「2」を取得できたことを確認できます。

- 次に、ストアド プロシージャを作成して、SCOPE_IDENTITY の結果を出力パラメーターで返すようにしてみましょう。

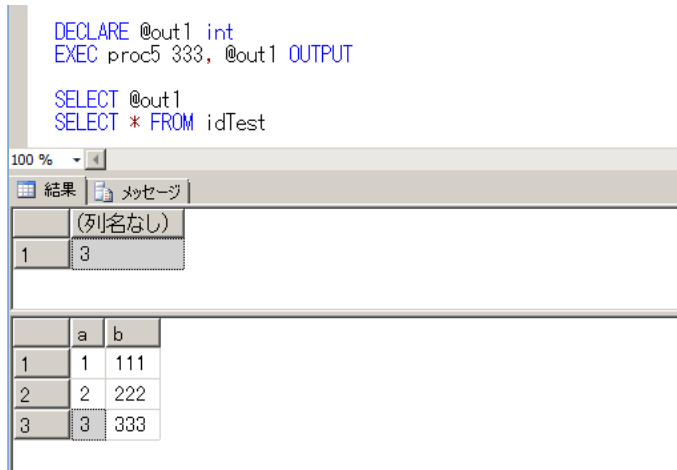
```
USE sampleDB
go
CREATE PROCEDURE proc5
    @p1 int
, @p2 int OUTPUT
AS
INSERT INTO idTest VALUES(@p1)
SELECT @p2 = SCOPE_IDENTITY()
```

- 続いて、ストアド プロシージャを実行してみましょう。

```
DECLARE @out1 int
EXEC proc5 333, @out1 OUTPUT

SELECT @out1

SELECT * FROM idTest
```



出力パラメーターから、最後に自動生成された値「3」を取得できたことを確認できます。

Note : ADO から出力パラメーターを取得するには SET NOCOUNT ON が必要

ADO.NET からではなく、VB 6.0 や VBA などの ADO から出力パラメーターを取得する場合は、次のようにストアード プロシージャの先頭に「SET NOCOUNT ON」を記述する必要があります。

```

CREATE PROCEDURE proc5
    @p1 int
    , @p2 int OUTPUT
AS
SET NOCOUNT ON
INSERT INTO idTest VALUES (@p1)
SELECT @p2 = SCOPE_IDENTITY()

```

ADO の場合は、SET NOCOUNT ON を付けない場合は、出力パラメーターを取得することができません。SET NOCOUNT ON は、ステートメントが実行されたときに発生する「～行処理されました」というメッセージを受け取らないという意味です（影響のあった行数を Count しない=No Count という主旨のコマンドです）。

ADO の場合は、「～行処理されました」というメッセージ自体を ADO のレコード セット（Recordset）として受け取ってしまうので、SET NOCOUNT ON を付けて、このメッセージを取得しないようにすることで、出力パラメーターを取得できるようにしています。ADO を利用する場合は、ストアード プロシージャの先頭へ SET NOCOUNT ON を付けるのが“お約束”だと思っても問題ありません。

■ SET NOCOUNT ON によるパフォーマンス上のメリット

SET NOCOUNT ON は、「～行処理されました」メッセージを受け取らない分、パフォーマンスが良いというメリットもあります。したがって、ADO を利用する場合だけでなく、ADO.NET を利用する場合にも、ストアード プロシージャの先頭へ SET NOCOUNT ON を付けておくことをお勧めします。

Note : 同時に実行された場合の保証

SCOPE_IDENTITY 関数で取得した値（採番された値）は、複数のユーザーから同時に INSERT ステートメントが実行された場合にはどうなると思いますか？ 結論から言うと、手順 3 で作成したストアード プロシージャのように、ストアード プロシージャ内で SCOPE_IDENTITY 関数を実行している場合は、複数のユーザーから同時に実行されたとしても、他のユーザーによって生成された IDENTITY 値を取得することはありません。自分が生成した最新の値を取得するので、安心して利用することができます。

SCOPE_IDENTITY 関数は、“同一モジュール”内であれば、一貫して同じ値（自分が追加した IDENTITY 値）を返すように作られているためです。モジュールは、「バッチ」または「ストアード プロシージャ」、「ユーザー定義関数」、「トリガー」を指します。

Note : IDENTITY は完全な連番ではない

IDENTITY プロパティは、厳密には完全な連番を作れないケースがあります。次の Step で説明する「トランザクション」内で生成された IDENTITY 値は、もしロールバックされた場合は、抜け番が発生する可能性があるためです。したがって、完全な連番を作成したい場合は、連番管理テーブルを用意するなど、自分で作成する必要があります。

Note : 旧バージョンの @@IDENTITY との比較

SCOPE_IDENTITY 関数は SQL Server 2000 からの機能で、それよりも前のバージョンの SQL Server では「@@IDENTITY」という関数を利用して IDENTITY 値を取得していました。しかし、@@IDENTITY では、トリガー内で生成された IDENTITY 値を取得してしまうという欠点がありました。したがって、IDENTITY 値の取得には、@@IDENTITY ではなく、SCOPE_IDENTITY を利用することをお勧めします。

3.8 RETURN コード

➤ RETURN コード

Step 3.2 で利用した **RETURN** ステートメントでは、ストアド プロシージャを（強制）終了する際に、RETURN コード（終了コード）を返すことができます。これは次のように利用します。

```
RETURN (整数値)
```

整数値を指定しない場合は、0 が返ります。

RETURN コードを取得する場合は、次のように記述します。

```
DECLARE @変数 int
EXEC @変数 = ストアドプロシージャ名
```

ストアド プロシージャの名前の前へ「@変数=」を付けることで RETURN コードを取得することができます。

➤ Let's Try

それでは、これを試してみましょう。

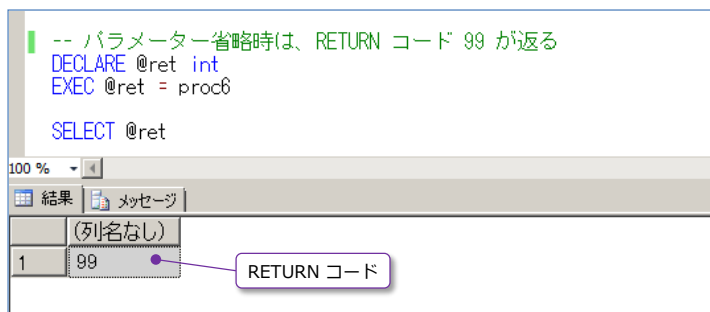
1. Step 3.2 で作成した proc1 ストアド プロシージャとほとんど同じですが、パラメーターが未入力の場合に、RETURN コードとして「99」を返し、正常終了した場合に「0」を返すようにしてみましょう。

```
USE sampleDB
go
CREATE PROCEDURE proc6
  @param1 int = NULL
AS
  IF @param1 IS NULL
  BEGIN
    PRINT 'パラメーター未入力!'
    RETURN (99)
  END

  SELECT * FROM emp WHERE deptno = @param1
  RETURN (0)
```

2. 次に、パラメーターを省略してストアド プロシージャを実行し、RETURN コードを取得してみましょう。

```
DECLARE @ret int
EXEC @ret = proc6
SELECT @ret
```

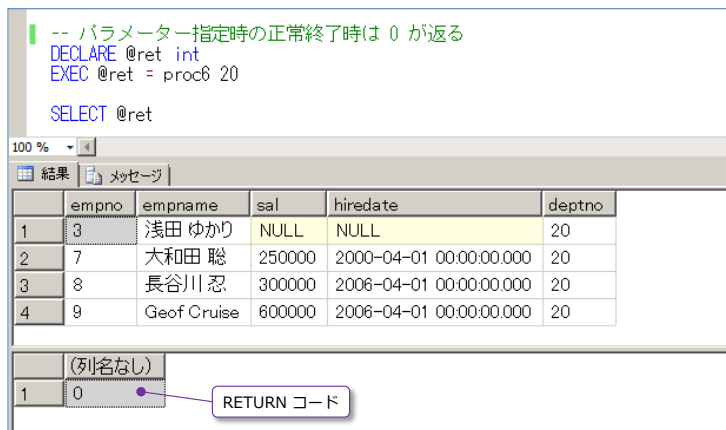
RETURN コードとして「99」を取得できたことを確認できます。

3. 続いて、パラメーターへ「20」を指定して、ストアド プロシージャを実行してみましょう。

```

DECLARE @ret int
EXEC @ret = proc6 20
SELECT @ret

```



今度は、RETURN コードとして「0」を取得できたことを確認できます。

Tips : ADO.NET で RETURN コードを取得する方法

ADO.NET (VB) から RETURN コードを取得するには、次のように **SqlParameter** クラスで **Direction** プロパティで **ReturnValue** を指定します。また、パラメーター名には @ を付ける必要はありません。

```

Dim p1 As SqlParameter = cmd.Parameters.Add("ret", SqlDbType.Int)
p1.Direction = ParameterDirection.ReturnValue

```

このように定義することで、RETURN コードを Value プロパティで取得できるようになります。ただし、出力パラメーターのときと同様、SqlDataReader を利用して結果を取得している場合は、それを Close (End Using) するまでは、Value プロパティを参照することができません。

3.9 ストアド プロシージャの削除

➤ ストアド プロシージャの削除

作成したストアド プロシージャを削除したい場合は、次のように「**DROP PROCEDURE**」ステートメントを使用します。

```
DROP PROCEDURE ストアドプロシージャ名
```

PROCEDURE の部分は、PROC と省略することもできます。

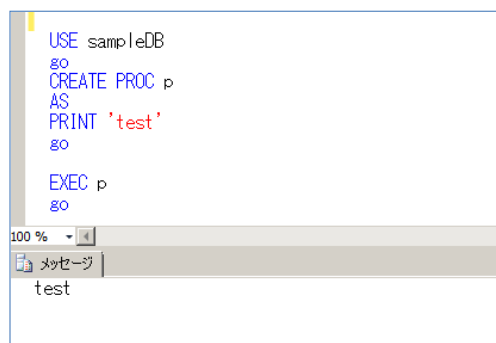
➤ Let's Try

それでは、これを試してみましょう。

1. まずは、次のように **sampleDB** データベース内へ「**p**」という名前のストアド プロシージャを作成します。

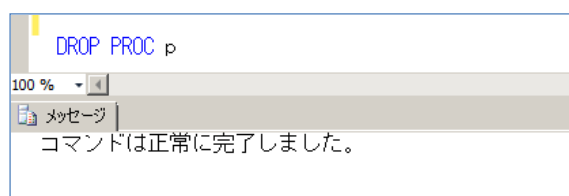
```
USE sampleDB
go
CREATE PROC p
AS
    PRINT 'test'
go

EXEC p
```



2. 次に、DROP PROCEDURE ステートメントを利用して、「**p**」ストアド プロシージャを削除してみましょう。

```
DROP PROC p
```



3.10 ストアド プロシージャの定義の表示

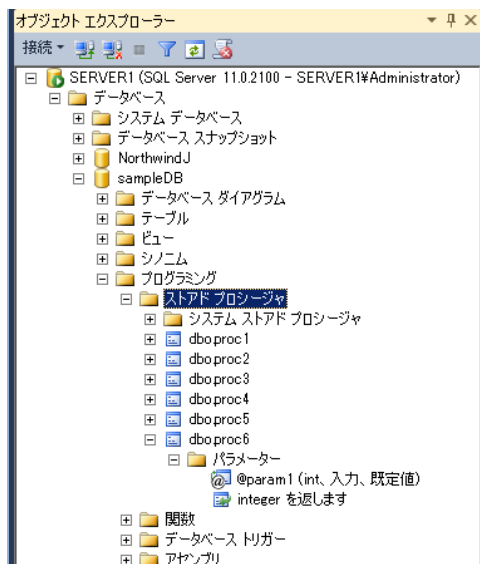
◆ ストアド プロシージャの定義の表示

作成したストアド プロシージャは、Management Studio のオブジェクト エクスプローラーを利用すると、簡単に定義を確認することができます。

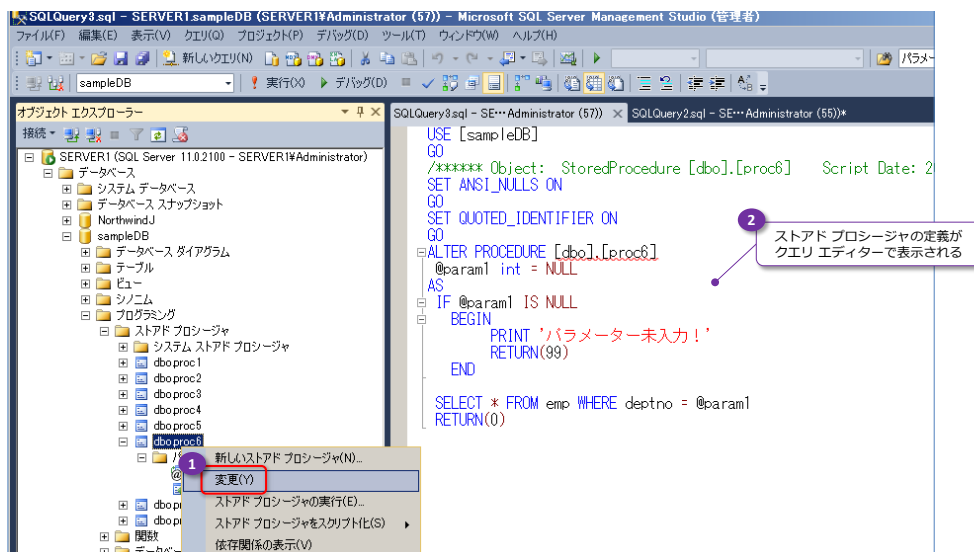
◆ Let's Try

それでは、これを試してみましょう。

1. まずは、Management Studio のオブジェクト エクスプローラーで、次のようにデータベース (sampleDB) 内の [プログラミング] から [ストアド プロシージャ] を展開します。



2. 次に、定義を確認したいストアド プロシージャを右クリックして、[変更] をクリックします。

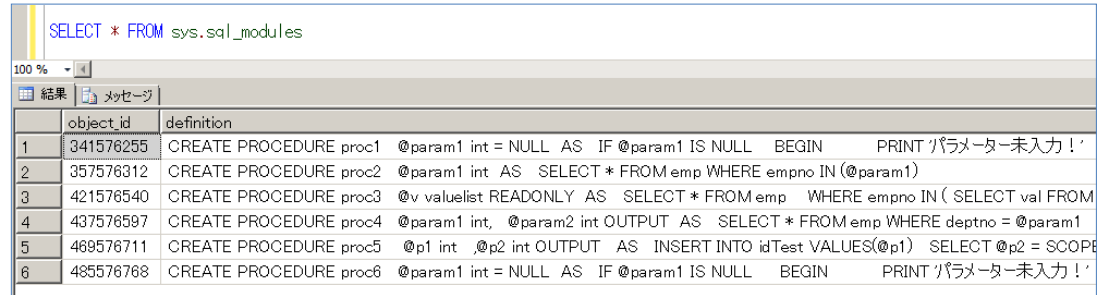


このようにクエリ エディターが開いて定義を確認することができ、また、ALTER

PROCEDURE ステートメントが自動生成されているので、定義を変更すれば、これを実行して変更を反映することもできるようになっています。

Note : sys.sql_modules で定義の確認

GUI 操作ではなく、SQL ステートメントから、ストアド プロシージャの定義を確認したい場合は、次のように sys.sql_modules システム ビューを参照することで、確認することができます。



| | object_id | definition |
|---|-----------|---|
| 1 | 341576255 | CREATE PROCEDURE proc1 @param1 int = NULL AS IF @param1 IS NULL BEGIN PRINT 'パラメーター未入力！' |
| 2 | 357576312 | CREATE PROCEDURE proc2 @param1 int AS SELECT * FROM emp WHERE empno IN (@param1) |
| 3 | 421576540 | CREATE PROCEDURE proc3 @v valuelist READONLY AS SELECT * FROM emp WHERE empno IN (SELECT val FROM |
| 4 | 437576597 | CREATE PROCEDURE proc4 @param1 int, @param2 int OUTPUT AS SELECT * FROM emp WHERE deptno = @param1 |
| 5 | 469576711 | CREATE PROCEDURE proc5 @p1 int ,@p2 int OUTPUT AS INSERT INTO idTest VALUES(@p1) SELECT @p2 = SCOPE |
| 6 | 485576768 | CREATE PROCEDURE proc6 @param1 int = NULL AS IF @param1 IS NULL BEGIN PRINT 'パラメーター未入力！' |

STEP 4. トランザクションとエラー処理

この STEP では、トランザクションとエラー処理について説明します。

この STEP では、次のことを学習します。

- ✓ トランザクションとは
- ✓ BEGIN TRAN、COMMIT TRAN
- ✓ 制約違反エラー時の動作
- ✓ 例外処理 (TRY ~ CATCH)
- ✓ エラーメッセージの取得 (ERROR_MESSAGE)
- ✓ THROW によるエラーの再スロー
- ✓ ユーザー定義エラー (RAISERROR)

4.1 トランザクションとは

✦ トランザクションとは

トランザクションは、データベース システムにおける処理の単位です。1 つまたは複数の SQL ステートメントを1つのトランザクションとしてデータベース サーバー (SQL Server) へ処理させることで、トランザクション (処理) の途中で障害が発生しても、データを一貫性のある状態に保つことができます。これは、“銀行の口座振り込み” を例に考えると分かりやすいと思います。次のように「A社の口座から、B社の口座に50万円振り込む」という処理があったとします。

口座テーブル

| 口座番号 | 口座名義 | 残高 |
|------|------|------|
| 100 | A 社 | 100万 |
| 101 | B 社 | 100万 |
| ⋮ | ⋮ | ⋮ |

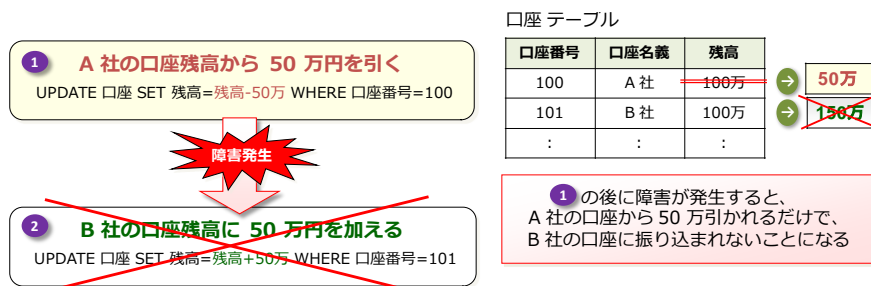
50万円
振り込む

これを実現するには、次の2つの SQL ステートメントが必要になります。

```
-- ① A社の残高から振り込み金額を引く
UPDATE 口座 SET 残高 = 残高 - 50万円 WHERE 口座番号 = 100

-- ② B社の残高に振り込み金額を加算する
UPDATE 口座 SET 残高 = 残高 + 50万円 WHERE 口座番号 = 101
```

銀行振り込みは、人間 (ユーザー) から見れば 1つの処理ですが、SQL Server にとっては複数のデータ操作が必要になります。このとき、トランザクションという機能がなかった場合に、処理の途中で障害が発生した場合を考えてみてください。次のように、「A社の口座残高から50万円引く」という1つ目の処理が終了した後に、不慮の障害 (停電や CPU 障害、メモリ障害など) が発生して、マシンがダウンしたとします。

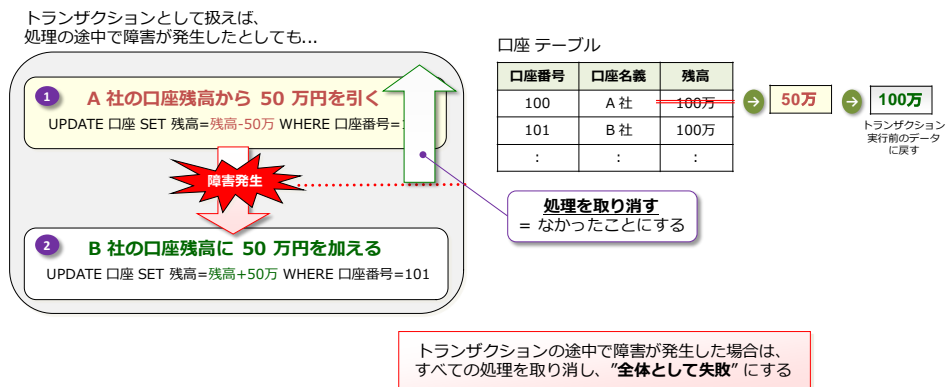


このままでは、振り込み処理が完了せず、A社の口座から50万円引かれただけになってしまいます (50万円が紛失しています)。このような事態を回避してくれる機能がトランザクションです。

✦ トランザクションの役割

トランザクションは、トランザクション内の処理が “すべて成功” か “すべて失敗” か (All or Nothing) を保証してくれる機能です。これにより、トランザクションの途中で障害が発生したとしても、すべての処理を “失敗” として扱うことで、処理が中途半端なままで終わることを回避す

ることができます。銀行振り込みの例では、「A社の口座から50万円引いた」という処理を失敗として扱えば（処理を取り消して、なかったことにすれば）、中途半端な状態を回避することができます。



このようにトランザクション機能を利用すると、処理の途中で障害が発生したとしても、すべての処理を取り消して、“全体として失敗”にできるようになります。

➡ ロールバック (Rollback) とコミット (Commit)

トランザクションでは、すべての処理を取り消す（トランザクションの開始時点まで戻す）ことを「**ロールバック**」(ROLLBACK)、すべての処理が完了することを「**コミット**」(COMMIT)と言います。

➡ トランザクションの実装

SQL Server では、次の 2 つをトランザクションとして扱います。

- **UPDATE、INSERT、DELETE などのデータ更新系のステートメント**

UPDATE や INSERT、DELETE などのデータ更新系のステートメントは、“そのステートメント単体”でトランザクションとして扱われます。

- **BEGIN TRANSACTION から COMMIT TRANSACTION で挟まれたステートメント**

複数のステートメントをトランザクションとして扱いたい場合は、ステートメントを「**BEGIN TRANSACTION**」ステートメントと「**COMMIT TRANSACTION**」ステートメントで挟みます。したがって、前述の銀行振り込みの例は、次のように記述することでトランザクションとして扱うことができます。

```
BEGIN TRANSACTION
UPDATE 口座 SET 残高 = 残高 - 50万円 WHERE 口座番号 = 100
UPDATE 口座 SET 残高 = 残高 + 50万円 WHERE 口座番号 = 101
COMMIT TRANSACTION
```

なお、TRANSACTION は、「**TRAN**」と省略することもできます。また、明示的にトランザクションを取り消すための「**ROLLBACK TRANSACTION**」というステートメントも用意さ

れています。

Note : Oracle や DB2 でのトランザクションの実装

Oracle や DB2 でのトランザクションの実装は、SQL Server とは異なり、BEGIN TRANSACTION というステートメントは存在しません。Oracle や DB2 の場合は、ステートメントが実行されると、そこから“自動的に”トランザクションが開始されて、COMMIT または ROLLBACK ステートメントが実行されるまでがトランザクションとして扱われます。

このようなトランザクションは、「暗黙的なトランザクション」(Implicit Transaction)とも呼ばれます。逆に、SQL Server のように BEGIN TRANSACTION ステートメントを利用して明示的にトランザクションを開始するタイプは、「明示的なトランザクション」とも呼ばれます。

なお、SQL Server を暗黙的なトランザクション モードとして動作させたい場合は、次のように SET コマンドを実行します。

SET IMPLICIT_TRANSACTIONS ON

このコマンドは、SQL Server に接続している間、または設定を OFF にするまで有効です。

➡ Let's Try

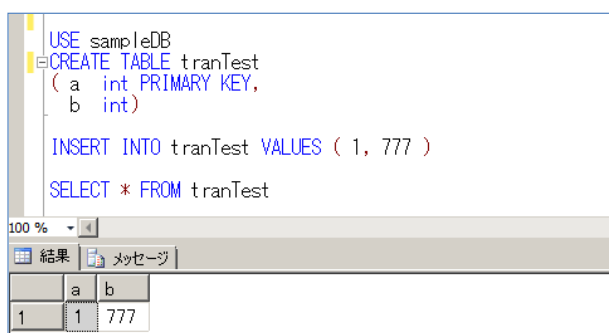
それでは、トランザクションを試してみましょう。

1. まずは、**sampleDB** データベース内へ「**tranTest**」という名前のテーブルを作成し、データを 1 件追加しておきます。

```
USE sampleDB
CREATE TABLE tranTest
( a int PRIMARY KEY,
  b int)

INSERT INTO tranTest VALUES ( 1, 777 )

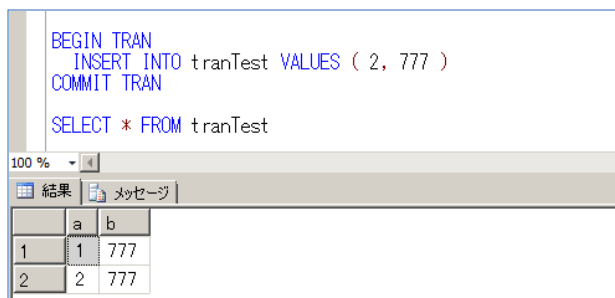
SELECT * FROM tranTest
```



| | a | b |
|---|---|-----|
| 1 | 1 | 777 |

2. 次に、データを追加するときに、BEGIN TRAN と COMMIT TRAN ステートメントで挟んで実行してみます。

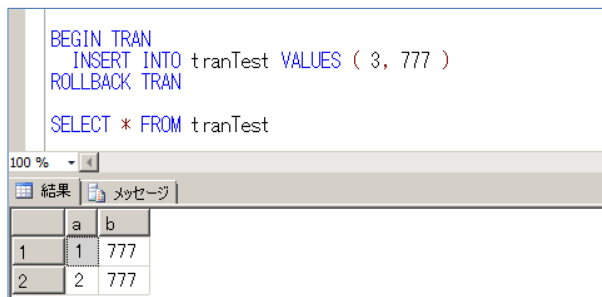
```
BEGIN TRAN
  INSERT INTO tranTest VALUES ( 2, 777 )
COMMIT TRAN
SELECT * FROM tranTest
```

3. 続いて、COMMIT TRAN の部分を ROLLBACK TRAN へ変更して、別の値を追加してみます。

```
BEGIN TRAN
INSERT INTO tranTest VALUES ( 3, 777 )
ROLLBACK TRAN

SELECT * FROM tranTest
```

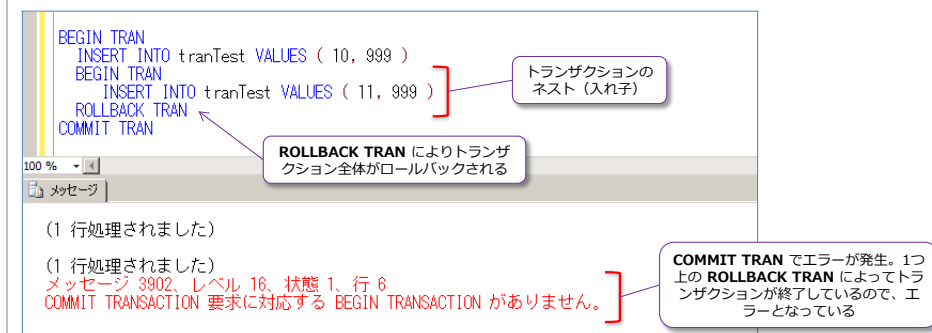


今度は、データが追加されずに、ロールバック（取り消し）されたことを確認できます。

このように SQL Server では、BEGIN TRAN ～ COMMIT TRAN または ROLLBACK TRAN を利用することでトランザクションの範囲を記述することができます。BEGIN TRAN ～ COMMIT TRAN で囲んだ範囲は、前述したように処理の途中で不慮の障害（停電や CPU 障害、メモリ障害など）が発生して、マシンがダウンしたとしても、処理の取り消し（ロールバック）を自動的に行ってくれます。

Note：ネストしたトランザクションの ROLLBACK

トランザクションはネスト（入れ子）にすることも可能です。この場合は、ネストしたトランザクション側で ROLLBACK TRAN が実行されると、トランザクション全体がロールバックされることに注意する必要があります。これは次のような状況です。



4.2 制約違反エラー時の動作

✦ 制約違反エラー時の動作

トランザクションは、不慮の障害（停電や CPU 障害、メモリ障害など）や、致命的なエラーが発生した場合には、ロールバックを行ってくれますが、制約違反エラーなどのステートメント エラーが発生した場合には、ロールバックを行ってくれません。この場合は、ロールバックを明示的に記述する必要があります。

✦ Let's Try

それでは、これを試してみましょう。前の Step で作成した **tranTest** テーブルは、次のように PRIMARY KEY（主キー制約）を設定していたので、これを利用して制約違反エラー時の動作を試します。

```
CREATE TABLE tranTest
( a  int PRIMARY KEY,
  b  int)

SELECT * FROM tranTest
```

| a | b |
|---|-----|
| 1 | 777 |
| 2 | 777 |

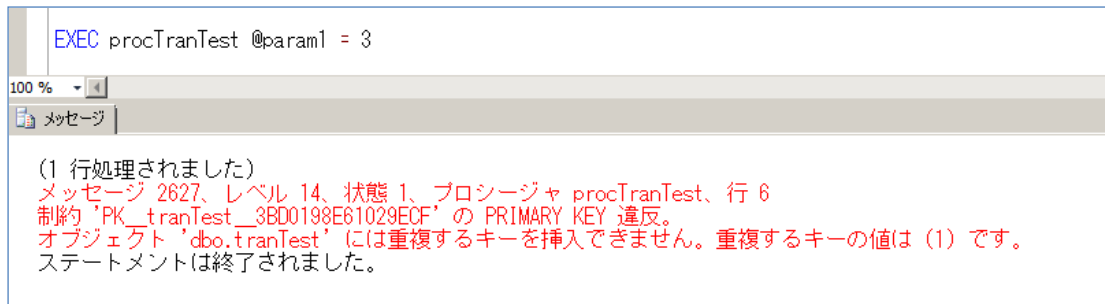
1. まずは、次のようにストアド プロシージャを作成して、意図的に PRIMARY KEY 制約違反エラーが発生するようにします。

```
USE sampleDB
go
CREATE PROCEDURE procTranTest
  @param1 int
AS
BEGIN TRAN
  INSERT INTO tranTest VALUES ( @param1, 999 )
  INSERT INTO tranTest VALUES ( 1, 999 )
COMMIT TRAN
```

2 つ目の INSERT ステートメントでは、a 列へ「1」を追加しようとしているので、ここで PRIMARY KEY 制約違反エラーが発生します。1 つ目の INSERT ステートメントは、入力パラメーター @param1 の値が既存の a 列のデータ（1 または 2）と重複しなければ正常終了します。

2. 次に、@param1 へ「3」を指定して、ストアド プロシージャを実行してみましょう。

```
EXEC procTranTest @param1 = 3
```



「1 行処理されました」と表示された後、エラー番号「**2627**」の PRIMARY KEY 制約違反エラーが発生していることを確認できます。

3. 次に、SELECT ステートメントを実行して、実際のデータを確認してみましょう。

```
SELECT * FROM tranTest
```

SELECT * FROM tranTest

100 %

結果 メッセージ

| | a | b |
|---|---|-----|
| 1 | 1 | 777 |
| 2 | 2 | 777 |
| 3 | 3 | 999 |

結果には、「**3**」のデータが追加され、エラーが発生したにも関わらず、トランザクションがコミットされてしまっていることを確認できます。

このように制約違反エラーなどのステートメント エラーの場合は、障害とは見なされず、ロールバックが発生しません。ステートメント エラー時にもロールバックされるようにするには、「**SET XACT_ABORT ON**」を追加するか、「**例外処理**」（エラー処理）を追加して、手動でロールバックを追加する必要があります。

4.3 SET XACT_ABORT ON の追加

➤ SET XACT_ABORT ON の追加

SET XACT_ABORT オプションは、既定では **OFF** に設定されていますが、これを **ON** へ設定することで、ステートメント エラーが発生した場合にも、トランザクションをロールバックできるようになります。**XACT** は「**Transaction**」の略、**Abort** は「**中止**」、「**中断**」という意味なので、(ステートメント エラー時の) トランザクションのロールバックを有効化するオプションです。

➤ Let's Try

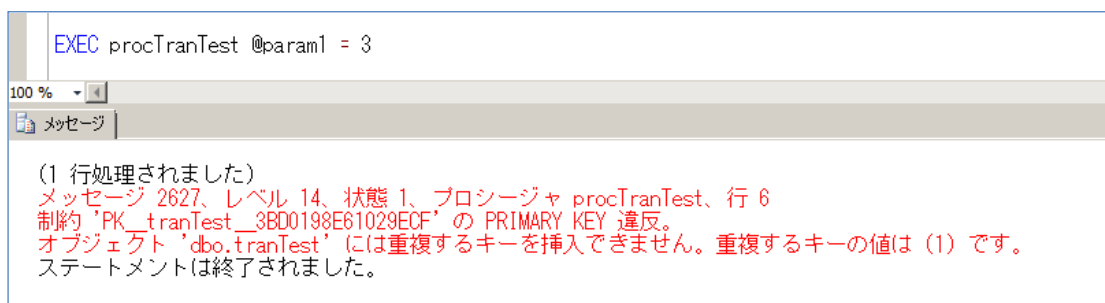
それでは、これを試してみましょう。

1. 前の Step で作成したストアード プロシージャ「**procTranTest**」へ「**SET XACT_ABORT ON**」を追加してみましょう。

```
USE sampleDB
go
ALTER PROCEDURE procTranTest
    @param1 int
AS
SET XACT_ABORT ON
BEGIN TRAN
    INSERT INTO tranTest VALUES ( @param1, 999 )
    INSERT INTO tranTest VALUES ( 1, 999 )
COMMIT TRAN
SET XACT_ABORT OFF
```

2. 次に、@param1 へ「4」を指定して、ストアード プロシージャを実行してみましょう。

```
EXEC procTranTest @param1 = 4
```



前の Step と同じエラーが発生していることを確認できます。

3. 実行後、テーブルの中身を参照します。

```
SELECT * FROM tranTest
```

SELECT * FROM tranTest

100 %

結果 メッセージ

| | a | b |
|---|---|-----|
| 1 | 1 | 777 |
| 2 | 2 | 777 |
| 3 | 3 | 999 |

4 は追加されていない

結果には、「4」が追加されていないことから、ロールバックがきちんと行われたことを確認できます。

このように **SET XACT_ABORT** オプションを **ON** へ設定しておけば、制約違反エラーが発生しても、トランザクションをロールバック（すべて取り消し）できるようになります。このオプションは、**SET XACT_ABORT OFF** を実行するか、接続が切れるまで有効です。

4.4 例外処理 : TRY ~ CATCH

✦ TRY ~ CATCH

Transact-SQL には、VB や C# で馴染みの TRY ~ CATCH を利用した例外処理機構も用意されています。構文は、次のとおりです。

```
BEGIN TRY
    例外の発生を調べるコード
END TRY
BEGIN CATCH
    例外が発生したときに処理するコード
END CATCH
```

BEGIN TRY から END TRY の範囲へ記述したステートメントで例外（エラー）が発生した場合は、BEGIN CATCH へジャンプして、END CATCH までの範囲へ記述したステートメントを実行できるようになります。

✦ Let's Try

それでは、これを試してみましょう。

1. 前の Step で作成したストアド プロシージャ「**procTranTest**」から、「**SET XACT_ABORT ON**」を削除して、「**TRY ~ CATCH**」を追加し、制約違反エラーが発生したときにロールバックが実行されるようにしてみましょう、

```
USE sampleDB
go
ALTER PROCEDURE procTranTest
    @param1 int
AS
BEGIN TRY
    BEGIN TRAN
        INSERT INTO tranTest VALUES ( @param1, 999 )
        INSERT INTO tranTest VALUES ( 1, 999 )
    COMMIT TRAN
END TRY
BEGIN CATCH
    ROLLBACK TRAN
END CATCH
```

2. 次に、@param1 へ「**4**」を指定して、ストアド プロシージャを実行してみましょう。

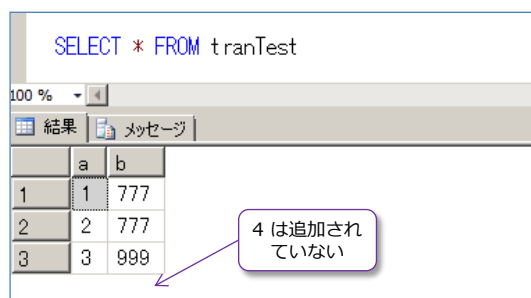
```
EXEC procTranTest @param1 = 4
```



今度は制約違反エラーが表示されていないことを確認できます。

3. 続いて、データが追加されたかどうかを確認しておきましょう。

```
SELECT * FROM tranTest
```



結果には、「4」のデータが追加されておらず、トランザクションがロールバックされていたことを確認できます。このように TRY ~ CATCH 構文を利用して、制約違反エラーが発生した場合に、ロールバックさせることができるようになります。

Note : ROLLBACK TRAN の記述し忘れに注意

CATCH ブロック内で、ROLLBACK TRAN を記述していない場合は、トランザクションがコミットもロールバックもされない状態（トランザクションが完了していない状態）で残ってしまうことに注意する必要があります。

この場合は、COMMIT または ROLLBACK TRAN が実行されるか、接続が切られるまで、トランザクション中の状態が続いてしまいます（自動的にロールバックしてくれるわけではありません）。VB や C# などのアプリケーションからストアド プロシージャを実行している場合は、アプリケーションを終了するまで、トランザクション中の接続が残ってしまいます。

このままでは、ロック待ちが解放されないデータが残ってしまうので、必ず CATCH ブロック内で、ROLLBACK TRAN を記述して、ロールバックを確実にしておくことが重要です。

Note : CATCH したエラーのアプリケーションへの通達

手順 2 の実行結果で、制約違反エラーが表示されなかったように、TRY ~ CATCH 構文を利用して、エラーをキャッチした場合は、エラーが吸収されてしまいます。したがって、アプリケーション側へエラーを通達するには、CATCH ブロック内でエラーを再スロー (throw) しなければなりません。これを行うには、後述の THROW または RAISERROR ステートメントを利用します。

4.5 エラー メッセージの取得： ERROR_MESSAGE

✦ エラー メッセージの取得

TRY ～ CATCH 構文の CATCH ブロックでは、エラー番号やエラー メッセージ、エラーの重大度レベルなど、エラーに関する周辺情報を取得することができます。具体的には、次のように“**ERROR_**”で始まるシステム関数を利用して取得することができます。

| 関数の名前 | 役割 |
|-----------------|-------------------------------|
| ERROR_NUMBER | エラー番号 |
| ERROR_MESSAGE | エラー メッセージ |
| ERROR_SEVERITY | エラーの重大度レベル |
| ERROR_STATE | エラーの状態番号 |
| ERROR_LINE | エラーが発生した行番号 |
| ERROR_PROCEDURE | エラーが発生したストアド プロシージャまたはトリガーの名前 |

✦ Let's Try

それでは、これを試してみましょう。

1. 前の Step で作成したストアド プロシージャ「**procTranTest**」の **CATCH** ブロックを次のように変更して、エラーに関する情報を取得してみましょう、

```
USE sampleDB
go
ALTER PROCEDURE procTranTest
    @param1 int
AS
BEGIN TRY
    BEGIN TRAN
        INSERT INTO tranTest VALUES ( @param1, 999 )
        INSERT INTO tranTest VALUES ( 1, 999 )
    COMMIT TRAN
END TRY
BEGIN CATCH
    ROLLBACK TRAN
    SELECT ERROR_NUMBER(), ERROR_MESSAGE(), ERROR_SEVERITY()
END CATCH
```

2. 次に、**@param1** へ「**4**」を指定して、ストアド プロシージャを実行してみましょう。

```
EXEC procTranTest @param1 = 4
```

| | | | |
|-------------------------------|--------|--|--------|
| EXEC procTranTest @param1 = 4 | | | |
| 100 % | | | |
| 結果 メッセージ | | | |
| | (列名なし) | (列名なし) | (列名なし) |
| 1 | 2627 | 制約 'PK_tranTest_3BD0198E61029ECF' の PRIMARY KEY 違反。オブジェ... | 14 |

ERROR_NUMBER 関数でエラー番号「2627」、**ERROR_MESSAGE** 関数でエラー メッセージ「PRIMARY KEY 違反」、**ERROR_SEVERITY** 関数でエラーの重大度レベル「14」を取得できていることを確認できます。

Note : CATCH できないエラー

TRY ~ CATCH では、キャッチできないエラーもあります。具体的には、エラーの重大度レベルが「10」以下のエラー（重大ではないエラーで、単なる情報メッセージに分類されるもの）と、重大度レベルが「20」以上のエラーのうち、再接続が不可となる深刻度の高いエラー（ハードウェア的な障害発生時などの致命的なエラー）の場合です。エラーの重大度レベルの詳細については、オンライン ブックの以下の場所を参考にしてください。

- SQL Server データベース エンジン
 - > テクニカル リファレンス
 - > エラーおよびイベントのリファレンス
 - > データベース エンジンのイベントとエラー
 - > データベース エンジン エラーについて
 - > データベース エンジン エラーの重大度

Note : SQL Server 2000 での例外処理

TRY ~ CATCH は、SQL Server 2005 からの機能で、SQL Server 2000 以前のバージョンには例外処理機構が備わっていませんでした。また、ERROR_MESSAGE 関数のようにエラー メッセージを取得する関数も用意されていませんでした。

SQL Server 2000 以前のバージョンでは、エラー処理は、次のように 1 つ 1 つのステートメントごとにエラーをチェックする「インライン エラー処理」として実装する必要がありました。

```
BEGIN TRAN
  INSERT INTO tranTest VALUES ( 4, 999 )
  IF @@ERROR <> 0
  BEGIN
    ROLLBACK TRAN
    RETURN
  END
  INSERT INTO tranTest VALUES ( 1, 999 )
  IF @@ERROR <> 0
  BEGIN
    ROLLBACK TRAN
    RETURN
  END
COMMIT TRAN
```

@@ERROR 関数は、直前のエラー番号を取得できるシステム関数で、直前のステートメントが成功した場合には "0" が格納されます。したがって、「IF @@ERROR <> 0」とすることで、エラーが発生したかどうかをチェックできるようになります。

なお、上記のエラー処理は、GOTO 構文を利用すると、次のように 1 つに集約させることも可能です。

```
BEGIN TRAN
  INSERT INTO tranTest VALUES ( 4, 999 )
  IF @@ERROR <> 0 GOTO err_label
  INSERT INTO tranTest VALUES ( 1, 999 )
  IF @@ERROR <> 0 GOTO err_label
COMMIT TRAN
RETURN

err_label:
  ROLLBACK TRAN
  RETURN
```

4.6 エラーの再スロー： THROW

✦ エラーの再スロー

TRY ～ CATCH を利用して例外処理を行っている場合は、エラーはアプリケーションには通達されず、CATCH 部の処理のみが実行されます（例外処理を行っていない場合は、エラーはアプリケーションにそのまま通達されます）。したがって、このままでは、アプリケーション側は正常に終了したのか、失敗したのかが判断できないため、アプリケーションに何かしらを通達する必要があります。これを行う手段が「**THROW** によるエラーの再スロー」または「**ユーザー定義エラーによる再スロー**」です。

✦ THROW によるエラーの再スロー

THROW は、SQL Server 2012 から提供されたステートメントで、CATCH 部でエラーを再スロー（エラーを再発生）することができる機能です。

それでは、これを試してみましょう。

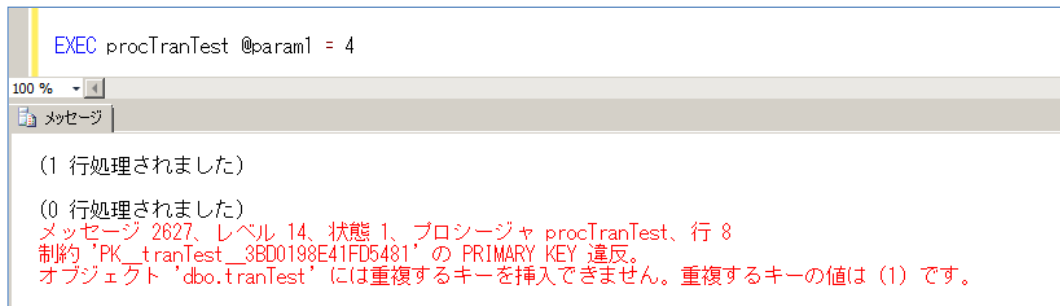
1. 前の Step で作成したストアード プロシージャ「**procTranTest**」の **CATCH** ブロックを次のように変更して、エラーを再スローしてみましょう、

```
USE sampleDB
go
ALTER PROCEDURE procTranTest
    @param1 int
AS
SET XACT_ABORT ON
BEGIN TRY
    BEGIN TRAN
        INSERT INTO tranTest VALUES ( @param1, 999 )
        INSERT INTO tranTest VALUES ( 1, 999 )
    COMMIT TRAN
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRAN
END CATCH
SET XACT_ABORT OFF
```

CATCH 部で **THROW** と記述することで、再度同じエラーを発生させることができます。しかし、制約違反エラーの場合にはロールバックが行われないので（**THROW** は全く同じエラーの再発生を行うだけなので）、**SET XACT_ABORT** オプションを **ON** へ設定するようにしています（これを設定しないとロールバックもコミットもされない未完了のトランザクションが残ってしまいます）。

2. 次に、**@param1** へ「4」を指定して、ストアード プロシージャを実行してみましょう。

```
EXEC procTranTest @param1 = 4
```



エラーが再スローされていることを確認できます。

このように **THROW** と **SET XACT_ABORT ON** を利用することで、エラーをアプリケーションへ通達できるようになります。

4.7 ユーザー定義エラー（RAISERROR）

✦ ユーザー定義エラー（RAISERROR）

アプリケーションへエラーを通知するためのもう 1 つの方法が「ユーザー定義エラー」による再スローです。ユーザー定義エラーは、**RAISERROR** というステートメントで発生させることができ、次のように利用します。

```
RAISERROR ( 'エラーメッセージ', エラー重大度, エラーの状態 )
[ WITH LOG ]
```

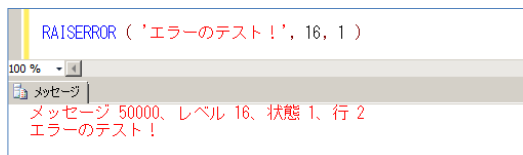
エラー重大度レベルには、「16」がユーザー定義エラー用として空いているので、これを利用することができます。**WITH LOG** を記述した場合は、Windows のイベント ログ（アプリケーションログ）へエラーを記録することができます（WITH LOG を記述しない場合は、イベント ログへの記録は行われません）。

✦ Let's Try

それでは、ユーザー定義エラーを試してみましょう。

1. まずは、次のように記述して、単純なエラーを発生させてみましょう。

```
RAISERROR ( 'エラーのテスト!', 16, 1 )
```



エラー番号は自動的に「50000」が割り当てられて、重大度レベル「16」のエラーを発生させることができたことを確認できます。

✦ ユーザー定義エラーによる再スロー

THROW は、SQL Server 2012 から提供された機能なので、SQL Server 2008 R2 以前のバージョンを利用して、アプリケーションへエラーを通知するには、**RAISERROR** ステートメントを利用する必要があります。

これを行うには、**CATCH** ブロック内で **RAISERROR** ステートメントを利用して、同じエラーを再度発生させて、アプリケーションへエラーを通知するようにします。

1. エラーを再度発生させるには、次のように記述します。

```
USE sampleDB
go
```

```

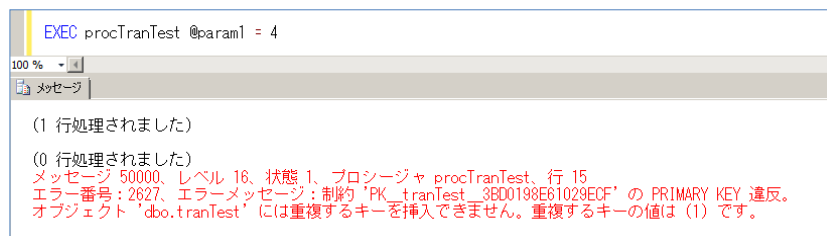
ALTER PROCEDURE procTranTest
    @param1 int
AS
BEGIN TRY
    BEGIN TRAN
        INSERT INTO tranTest VALUES ( @param1, 999 )
        INSERT INTO tranTest VALUES ( 1, 999 )
    COMMIT TRAN
END TRY
BEGIN CATCH
    ROLLBACK TRAN
    DECLARE @errMes varchar(1000)
    SELECT @errMes = 'エラー番号 : ' + CONVERT( varchar, ERROR_NUMBER() )
        + '、エラーメッセージ : ' + ERROR_MESSAGE()
    RAISERROR (@errMes, 16, 1)
END CATCH

```

ERROR_NUMBER 関数で取得したエラー番号と、**ERROR_MESSAGE** 関数で取得したエラーメッセージを文字列連結して、**RAISERROR** ステートメントの第 1 引数へ与えています。

2. 次に、@param1 へ「4」を指定して、ストアド プロシージャを実行してみましょう。

```
EXEC procTranTest @param1 = 4
```



エラー番号に「50000」が割り当てられたエラーへ、PRIMARY KEY 制約違反のエラー番号とエラー メッセージを取得できていることを確認できます。このようにエラーを再度発生させれば、アプリケーション側でエラーを取得できるようになります。

Note : ADO の場合は SET NOCOUNT ON が必要

ADO.NET ではなく、VB 6.0 や VBA など ADO を利用してアプリケーションを作成している場合は、ストアド プロシージャの先頭に「SET NOCOUNT ON」を記述しないとエラーを取得することができません。ADO の場合は、「n 行処理されました」というメッセージが生成されると、アプリケーションにエラーが通達されないという仕様があるためです。

Note : RAISERROR ステートメントで変換指定子の利用

RAISERROR ステートメントには、C 言語の printf 関数でのフォーマット指定子と同じように、エラー メッセージを動的に変更可能な変換指定子として「%d」（整数 : decimal）と「%s」（文字列 : string）などが用意されています。これを利用すると、上記のストアド プロシージャの RAISERROR ステートメントは、次のように記述することができます。

```

BEGIN CATCH
    ROLLBACK TRAN
    DECLARE @errNum int = ERROR_NUMBER()
    DECLARE @errMes varchar(1000) = ERROR_MESSAGE()
    RAISERROR ('エラー番号: %d、エラーメッセージ: %s', 16, 1, @errNum, @errMes)
END CATCH

```

変換指定子は、複数指定することができ、代入したい値を第4引数以降へ指定します。最初に指定した %d（整数）へは ERROR_NUMBER 関数で取得したエラー番号を代入して、2つ目に指定した %s（文字列）へは ERROR_MESSAGE 関数で取得したエラーメッセージを代入するようにしています。

このように変換指定子を利用すると、関数で取得した結果などを動的に文字列として生成することができるので、大変便利です。

Note：再スロー用のストアード プロシージャの作成

ユーザー定義エラーを利用したエラーの再スローは、ストアード プロシージャを作成しておく、ストアード プロシージャの呼び出しだけで済むようになるので便利です。たとえば、SQL Server 2008 R2 のオンライン ブックの以下の場所では、次のストアード プロシージャが紹介されています。

Transact-SQL での TRY...CATCH の使用

[http://msdn.microsoft.com/ja-jp/library/ms179296\(v=sql.105\).aspx](http://msdn.microsoft.com/ja-jp/library/ms179296(v=sql.105).aspx)

```

CREATE PROCEDURE usp_RethrowError AS
-- Return if there is no error information to retrieve.
IF ERROR_NUMBER() IS NULL
    RETURN;

DECLARE
    @ErrorMessage NVARCHAR(4000),
    @ErrorNumber INT,
    @ErrorSeverity INT,
    @ErrorState INT,
    @ErrorLine INT,
    @ErrorProcedure NVARCHAR(200);

-- Assign variables to error-handling functions that
-- capture information for RAISERROR.
SELECT
    @ErrorNumber = ERROR_NUMBER(),
    @ErrorSeverity = ERROR_SEVERITY(),
    @ErrorState = ERROR_STATE(),
    @ErrorLine = ERROR_LINE(),
    @ErrorProcedure = ISNULL(ERROR_PROCEDURE(), '-');

-- Build the message string that will contain original
-- error information.
SELECT @ErrorMessage =
    N'Error %d, Level %d, State %d, Procedure %s, Line %d, ' +
    'Message: ' + ERROR_MESSAGE();

-- Raise an error: msg_str parameter of RAISERROR will contain
-- the original error information.
RAISERROR
(
    @ErrorMessage,
    @ErrorSeverity,
    1,
    @ErrorNumber, -- parameter: original error number.
    @ErrorSeverity, -- parameter: original error severity.
    @ErrorState, -- parameter: original error state.
    @ErrorProcedure, -- parameter: original error procedure name.
    @ErrorLine -- parameter: original error line number.
);

```

✦ エラーの登録： sp_addmessage

何度も呼び出すエラー メッセージは、**sp_addmessage** システム ストアド プロシージャを利用して、登録しておくことができます。構文は、次のとおりです。

```
sp_addmessage エラー番号, 重大度レベル, エラーメッセージ, 言語, ログへ記録の有無
```

エラー番号（メッセージ番号）には、**50001～2,147,483,647** までの整数を指定し、重大度レベルには、前述したように「**16**」がユーザー定義エラー用として空いています。第 4 引数の言語（エラー メッセージ内容を記述する言語）には、'**us_english**' を指定したものを追加した後、各言語に対応したもの（日本語の場合は '**japanese**' を指定）を追加します。第 5 引数は、Windows のイベント ログへ記録するかどうかを '**TRUE**' または '**FALSE**' で指定し、省略時は '**FALSE**'（記録しない）が設定されます。

sp_addmessage で登録したエラー メッセージは、次のように **RAISERROR** ステートメントの第 1 引数でエラー番号を指定することで呼び出すことができます。

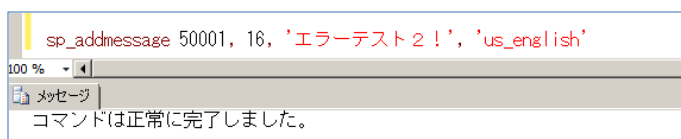
```
RAISERROR ( エラー番号, 重大度レベル, 状態 )
```

✦ Let's Try

それでは、**sp_addmessage** を試してみましょう。

1. 次のように記述して、エラー番号が「**50001**」のエラー メッセージを登録してみます。

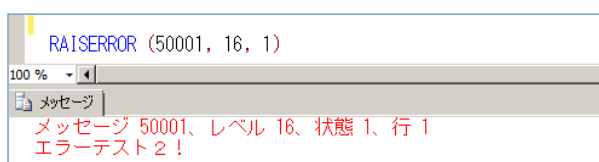
```
sp_addmessage 50001, 16, 'エラーテスト 2 !', 'us_english'
```



言語に **us_english** を指定した場合は、本来は英語でエラー メッセージを記述する必要がありますが、日本語で登録することも可能です。また、本来は日本語のエラー メッセージでは、'**Japanese**' を指定して登録する必要がありますが、ここでは省略します。

2. 続いて、**RAISERROR** ステートメントを利用して、登録したエラーを呼び出してみましょう。

```
RAISERROR (50001, 16, 1)
```



登録したエラー メッセージを取得できたことを確認できます。

登録したエラーの一覧を取得： sys.messages

登録したエラーの一覧を取得したい場合は、**sys.messages** システム ビューを参照します。

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

-- エラー一覧
SELECT * FROM sys.messages

| | message_id | language_id | severity | is_event_logged | text |
|---|------------|-------------|----------|-----------------|---|
| 1 | 50001 | 1033 | 16 | 0 | エラーテスト2！ |
| 2 | 21 | 1033 | 20 | 0 | Warning: Fatal error %d occurred at %S_DATE. Note t... |
| 3 | 101 | 1033 | 15 | 0 | Query not allowed in Waitfor. |
| 4 | 102 | 1033 | 15 | 0 | Incorrect syntax near '%*ls'. |
| 5 | 103 | 1033 | 15 | 0 | The %S_MSG that starts with '%*ls' is too long. Maxi... |

登録したエラーの変更と削除

登録したエラーを変更したい場合は、**sp_addmessage** システム ストアド プロシージャの第 6 引数で 'REPLACE' を指定します。これは次のように利用します。

```
sp_addmessage 50001, 16, 'エラーの変更', 'us_english', 'FALSE', 'REPLACE'
```

登録したエラーを削除したい場合は、**sp_dropmessage** システム ストアド プロシージャを次のように利用します。

```
sp_dropmessage 50001, 'us_english'
```

Note：変換指定子を利用する場合の注意事項

前述の Note で紹介した変換指定子として **%d** (整数) と **%s** (文字列) を利用する方法は、言語へ 'us_english' を指定する場合は、同じように利用することができます。

```
-- 変数指定子
EXEC sp_addmessage 50001, 16, 'テスト %s テスト %d', 'us_english'
RAISERROR (50001, 16, 1, '文字列', 999)
```

メッセージ 50001、レベル 16、状態 1、行 2
テスト 文字列 テスト 999

しかし、言語へ 'japanese' などの他の言語を指定した場合は、**%d** と **%s** は利用できなくなります。代わりに、**%1!** と **%2!**、**%3!**、… (変換する順番を指定した連続番号に ! を付けたもの) を指定する必要があります。したがって、上記の例は、'japanese' の場合は、次のように記述する必要があります。

```
-- 変数指定子 japanese の場合
EXEC sp_addmessage 50001, 16, 'japanese テスト %1! テスト %2!', 'japanese'
RAISERROR (50001, 16, 1, '文字列', 999)
```

メッセージ 50001、レベル 16、状態 1、行 3
japanese テスト 文字列 テスト 999

Note : ADO.NET 2.0 でトランザクション : System.Transactions の TransactionScope

この Step では、Transact-SQL を利用したトランザクション制御を説明してきましたが、VB や C# などのアプリケーションからトランザクションを制御することもできます。ADO.NET 2.0 以降の場合には、

System.Transactions 名前空間の「**TransactionScope**」クラスを利用すると、簡単に実装することができます。これを利用するには、「**参照の追加**」から「**System.Transactions.dll**」を参照して、次のように記述します。

```
Imports System.Data.SqlClient
Imports System.Transactions
:
Using cn As New SqlConnection("Server=localhost;Database=sampleDB;Integrated Security=SSPI;")
Try
    Using tx As New TransactionScope()
        cn.Open()
        Using cmd As New SqlCommand()
            cmd.Connection = cn
            cmd.CommandText = "INSERT INTO tranTest VALUES(4, 999)"
            cmd.ExecuteNonQuery()
            cmd.CommandText = "INSERT INTO tranTest VALUES(1, 999)"
            cmd.ExecuteNonQuery()
            ' トランザクションのコミット
            tx.Complete()
        End Using
    End Using
Catch ex As System.Exception
    MessageBox.Show(ex.Message)
Finally
    cn.Close()
End Try
End Using
```

Using を利用して **TransactionScope** オブジェクトを作成し、**SqlConnection** が **Open** されると、自動的にトランザクションが開始されて、**Complete()** メソッド（コミットの合図）が呼ばれるまでの範囲をトランザクションとして扱うことができます。また、この間に例外（エラー）が発生した場合は、自動的にロールバックが実行されます。

■ 分離レベルが Serializable であることに注意

TransactionScope クラスは、デフォルトでは**分離レベル**（Isolation Level）が **Serializable** に設定されていることに注意する必要があります。この場合は、SELECT ステートメントが実行された場合に、共有ロックがトランザクションが完了するまで保持されてしまうので、同時実行性が大きく低下します。ロックについては、本自習書シリーズの「**ロックと読み取り一貫性**」で詳しく説明していますので、こちらもぜひご覧いただければと思います。。

共有ロックをすぐに解放するようにするには、分離レベルを **Read Committed** へ変更します。分離レベルを変更するには、**TransactionOptions** クラスを利用して次のように記述します。

```
' 分離レベルの変更
Dim txOp As New TransactionOptions
txOp.IsolationLevel = IsolationLevel.ReadCommitted
Using tx As New TransactionScope(TransactionScopeOption.Required, txOp)
    :
End Using
```

■ 複数 Connection では MSDTC サービスが利用される

TransactionScope では、複数の **Connection** が **Open** された場合には、自動的に分散トランザクションとして実行されて、**MSDTC**（Distributed Transaction Coordinator）サービスが利用されます（このサービスが停止している場合は、例外が発生します）。

Note : ADO.NET 1.1 でトランザクション : SqlTransaction

TransactionScope は、ADO.NET 2.0 からの機能なので、ADO.NET 1.1 でトランザクションを実装するには、**SqlTransaction** クラスを利用します。これは次のように記述します。

```
Imports System.Data.SqlClient
:
Using cn As New SqlConnection("Server=localhost;Database=sampleDB;Integrated Security=SSPI;")
    cn.Open()
    Using cmd As New SqlCommand()
        cmd.Connection = cn
        Dim tx As SqlTransaction = cn.BeginTransaction()
        cmd.Transaction = tx
        Try
            cmd.CommandText = "INSERT INTO tranTest VALUES(4, 999)"
            cmd.ExecuteNonQuery()
            cmd.CommandText = "INSERT INTO tranTest VALUES(1, 999)"
            cmd.ExecuteNonQuery()
            tx.Commit()
        Catch ex As System.Exception
            tx.Rollback()
            MessageBox.Show(ex.Message)
        Finally
            cn.Close()
        End Try
    End Using
End Using
```

SqlTransaction クラスでは、**BeginTransaction** メソッドでトランザクションを開始して、**Commit** メソッドでコミット、**Rollback** メソッドでロールバックを実行することができます。なお、SqlTransaction クラスの場合は、TransactionScope の場合とは異なり、分離レベルは **Read Committed** として実行されます。

STEP 5. その他

この STEP では、「オブジェクトの依存関係の表示機能」と、地理情報を扱うことができる「Spatial データ型」、Windows ファイルを格納できる「FileTable」機能について説明します。

この STEP では、次のことを学習します。

- ✓ オブジェクトの依存関係の表示
- ✓ Spatial データ型
- ✓ FileTable による Windows ファイルのサポート

5.1 オブジェクトの依存関係の表示

✦ オブジェクトの依存関係の表示

SQL Server 2008 からは、オブジェクト（ビューやストアド プロシージャ、ユーザー定義関数など）の依存関係を表示するための機能として、次の 3 つのビュー（関数）が追加されました。

- sql_expression_dependencies
- dm_sql_referencing_entities
- dm_sql_referenced_entities

これらのビューにより、ストアド プロシージャが依存しているテーブルや、ユーザー定義関数を利用しているオブジェクトを容易に把握できるようになったので、大変便利です。

✦ Let's Try

それでは、これを試してみましょう。

1. まずは、**sampleDB** データベースの「**emp**」テーブルをもとに、ビューやユーザー定義関数、ストアド プロシージャを作成しましょう。

```
USE sampleDB
go
-- ビュー empView の作成 (emp テーブルに依存)
CREATE VIEW dbo.empView
AS
SELECT empno, empname FROM emp
go

-- ユーザー定義関数 empFunc1 の作成
CREATE FUNCTION dbo.empFunc1 (@p1 int) RETURNS int
BEGIN
    RETURN @p1 * 100
END
go

-- ストアドプロシージャ empProc1 の作成 (empFunc1 と empView に依存)
CREATE PROC empProc1
AS
SELECT dbo.empFunc1 (empno) FROM emp
go

-- ストアドプロシージャ empProc2 の作成 (empProc1 に依存)
CREATE PROC empProc2
AS
EXEC empProc1
go
```

➤ sql_expression_dependencies

- 次に、**sql_expression_dependencies** カタログ ビューを利用して、作成したオブジェクトの依存関係を表示してみましょう。

```
SELECT OBJECT_NAME( referencing_id ), referenced_entity_name AS 依存元, *
FROM sys.sql_expression_dependencies
```

| | (列名なし) | 依存元 | referencing_id | referencing_minor_id | referencing_class | referencing_class_desc |
|----|--------------|-----------|----------------|----------------------|-------------------|------------------------|
| 1 | proc1 | emp | 341576255 | 0 | 1 | OBJECT_OR_COLUMN |
| 2 | proc2 | emp | 357576312 | 0 | 1 | OBJECT_OR_COLUMN |
| 3 | proc3 | emp | 421576540 | 0 | 1 | OBJECT_OR_COLUMN |
| 4 | proc3 | valuelist | 421576540 | 0 | 1 | OBJECT_OR_COLUMN |
| 5 | proc4 | emp | 437576597 | 0 | 1 | OBJECT_OR_COLUMN |
| 6 | proc5 | idTest | 469576711 | 0 | 1 | OBJECT_OR_COLUMN |
| 7 | proc6 | emp | 485576768 | 0 | 1 | OBJECT_OR_COLUMN |
| 8 | procTranTest | tranTest | 549576896 | 0 | 1 | OBJECT_OR_COLUMN |
| 9 | empView | emp | 581577110 | 0 | 1 | OBJECT_OR_COLUMN |
| 10 | empProc1 | emp | 613577224 | 0 | 1 | OBJECT_OR_COLUMN |
| 11 | empProc1 | empFunc1 | 613577224 | 0 | 1 | OBJECT_OR_COLUMN |
| 12 | empProc2 | empProc1 | 629577281 | 0 | 1 | OBJECT_OR_COLUMN |

sql_expression_dependencies ビューでは、**referenced_entity_name** 列を参照することで依存元となっているオブジェクトを取得することができます。**empView** が **emp** テーブルへ依存していること、**empProc1** ストアド プロシージャが **emp** テーブルと **empFunc1** ユーザー定義関数に依存していることを確認できたと思います。

➤ dm_sql_referenced_entities

dm_sql_referenced_entities 動的管理関数を利用すると、依存元の列名を取得することもできます。では、これを試してみましょう。

- 次のように記述して、**empView** ビューが依存している列名を取得してみます。

```
SELECT referenced_entity_name AS 依存元, referenced_minor_name AS 依存元の列名, *
FROM sys.dm_sql_referenced_entities ( 'dbo.empView', 'OBJECT' )
```

| | 依存元 | 依存元の列名 | referencing_minor_id | referenced_server_name | referenced_database_name |
|---|-----|---------|----------------------|------------------------|--------------------------|
| 1 | emp | NULL | 0 | NULL | NULL |
| 2 | emp | empno | 0 | NULL | NULL |
| 3 | emp | empname | 0 | NULL | NULL |

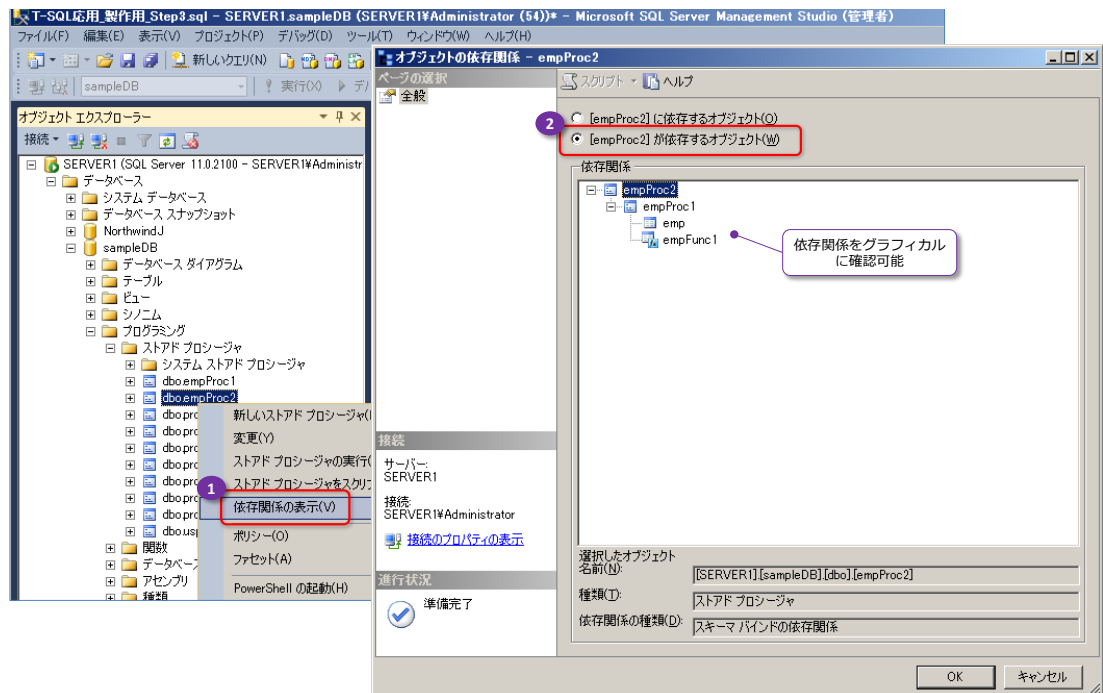
dm_sql_referenced_entities 関数では、第 1 引数へ依存関係を調べたいオブジェクト名

を指定することで、**referenced_minor_name** 列で依存元の列名を取得することができます。

◆ 依存関係を GUI で確認

オブジェクト間の依存関係は、Management Studio を利用して、グラフィカルに確認することもできます。では、これを試してみましょう。

1. 次のようにストアード プロシージャ「empProc2」を右クリックして、[依存関係の表示] をクリックします。



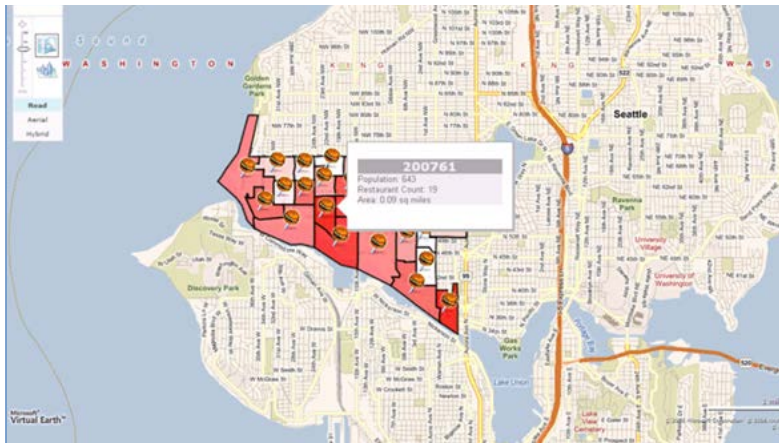
これにより、[オブジェクトの依存関係] ダイアログが表示されるので、「**empProc2 に依存するオブジェクト**」をクリックすると、依存関係をグラフィカルに確認することができます。

このように、SQL Server 2008 からは、オブジェクト間の依存関係を簡単に表示できるようになったので、大変便利です。

5.2 Spatial データ型による地図データのサポート

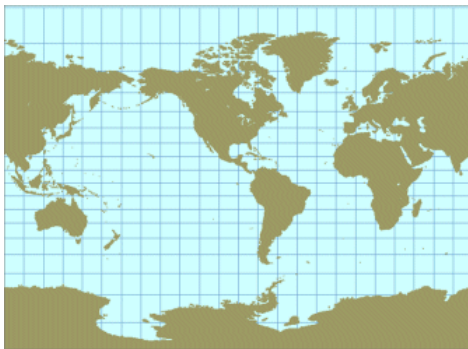
◆ Spatial データ型 (geometry、geography)

Spatial データ型は、GIS（地理情報システム）における地図データ（緯度と経度）を格納できるデータ型で、**Bing マップ**（旧 Virtual Earth）と連携して、次のように地図アプリケーションとして利用することができる機能です。



Spatial データ型には、**geometry データ型**と **geography データ型**の 2 種類があり、次のような違いがあります。

平面モデル (geometry データ型)



測地モデル (geography データ型)



geometry データ型は「**平面**」(2次元)として捕え、**geography データ型**は「**球体**」として捕えるという違いがあります。

それでは、これらのデータ型を試してみましょう。

◆ geometry データ型

まずは、geometry データ型を利用して、単純なデータの格納の仕方などを試してみましょう。次のように記述して、データを格納、取得してみます。

```
-- データベースの作成
CREATE DATABASE GeoTestDB
go
```

```
-- テーブルの作成。geom 列を geometry データ型へ設定
USE GeoTestDB
CREATE TABLE geomTest
( a int IDENTITY(1,1) PRIMARY KEY
, geom geometry )

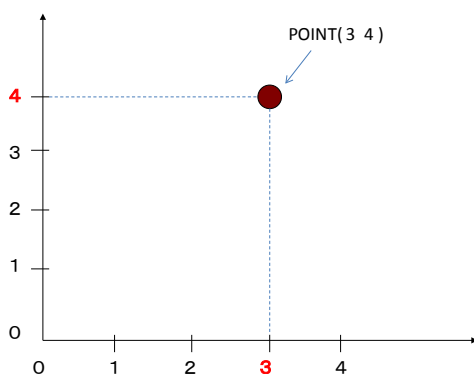
-- データの格納は STGeomFromText。POINT と指定することで「点」を追加
INSERT INTO geomTest
VALUES ( geometry::STGeomFromText(' POINT(3 4)', 0) )

-- POLYGON と指定することで「多角形」データを追加
INSERT INTO geomTest
VALUES ( geometry::STGeomFromText(' POLYGON((0 0, 2 0, 2 2, 0 2, 0 0))', 0) )

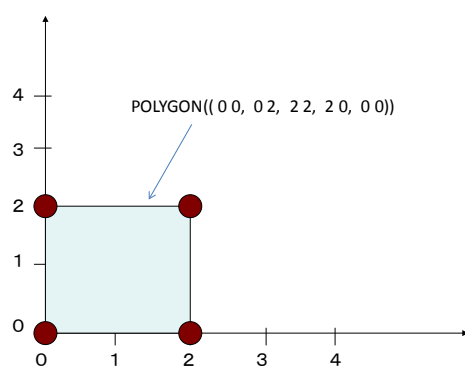
-- データの確認は STAsText
SELECT geom.STAsText(), * FROM geomTest
```

[illegible]

POINT は点



POLYGON は多角形



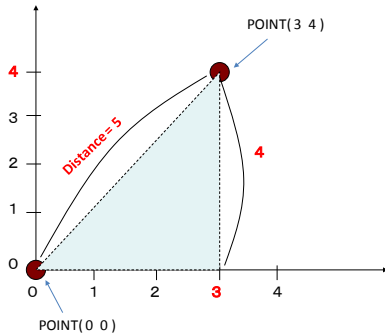
◆ STDistance (2点間の距離)

次に、STDistance 関数を利用して、2 点間 (0 0) と (3 4) の距離を取得してみましょう。

```
DECLARE @g1 geometry;
DECLARE @g2 geometry;
SET @g1 = geometry::STGeomFromText('POINT(0 0)', 0);
SET @g2 = geometry::STGeomFromText('POINT(3 4)', 0);
SELECT @g1.STDistance(@g2);
```

| | |
|---|--------|
| | (列名なし) |
| 1 | 5 |

結果は、5 が返りますが、次のような三角形を思い浮かべると、イメージが沸くのではないでしょうか。

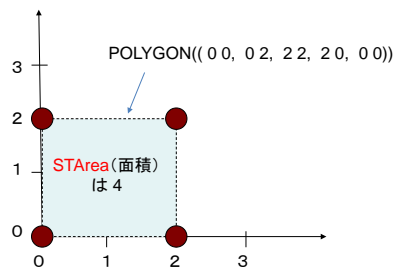


➤ STArea (多角形の面積)

次に、STArea 関数を利用して、面積を取得してみましょう。

```
-- STArea (面積)
DECLARE @g geometry;
SET @g = geometry::STGeomFromText(' POLYGON((0 0, 0 2, 2 2, 2 0, 0 0))', 0);
SELECT @g.STArea()
```

| | (列名なし) |
|---|--------|
| 1 | 4 |



このように Spatial データ型では、多角形などを処理するための特殊な関数が多数用意されています。その他の関数については、オンライン ブックの「**Transact-SQL リファレンス**」→「**データ型**」→「**空間型**」に記載されています。

➤ geography データ型と Bing マップ連携

次に、**geography** データ型を利用して、**Bing マップ**と連動したアプリケーションを作成してみましょう。具体的には、次のように「つくば市」内の 4 つの郵便局を Bing マップ上へ表示するようなアプリケーションを作成し、郵便局の「経度」と「緯度」を **geography** データ型の列へ格納し、それを ASP.NET Web フォームから取得するようにします。



まずは、4 つの郵便局のデータと TX（つくばエクスプレス）のつくば駅の経度と緯度を POINT（点）として、次のように「郵便局」テーブルへ格納します（この SQL は、サンプル スクリプト内の「Step5-2_Script_Bing.sql」というファイル名で置いてあります）。

```
USE GeoTestDB
go
CREATE TABLE 郵便局
( id int IDENTITY (1,1) PRIMARY KEY,
  郵便局名 varchar(100),
  住所      varchar(200),
  geog      geography )
go

INSERT INTO 郵便局(郵便局名, 住所, geog)
VALUES (' TXつくば駅', ' 茨城県つくば市吾妻2丁目 128 ',
        geography::STGeomFromText(' POINT(140.111561 36.082757)', 4612))

INSERT INTO 郵便局(郵便局名, 住所, geog)
VALUES (' 筑波学園郵便局', ' 茨城県つくば市吾妻1丁目 13-2 ',
        geography::STGeomFromText(' POINT(140.11575 36.082818)', 4612));

INSERT INTO 郵便局(郵便局名, 住所, geog)
VALUES (' 葛城郵便局', ' 茨城県つくば市荻間388 ',
```

```

geography::STGeomFromText(' POINT (140.09617 36.077463)', 4612));

INSERT INTO 郵便局 (郵便局名, 住所, geog)
VALUES (' 谷田部松代郵便局', ' 茨城県つくば市松代4丁目200-1',
        geography::STGeomFromText(' POINT (140.103989 36.063796)', 4612));

INSERT INTO 郵便局 (郵便局名, 住所, geog)
VALUES (' 小野川郵便局', ' 茨城県つくば市館野464',
        geography::STGeomFromText(' POINT (140.113192 36.043797)', 4612));

SELECT geog.STAsText(), * FROM 郵便局

```

| | (列名なし) | id | 郵便局名 | 住所 | geog |
|---|------------------------------|----|----------|-------------------|----------------------|
| 1 | POINT (140.111561 36.082757) | 1 | TXつくば駅 | 茨城県つくば市吾妻2丁目128 | 0x04120000010CEE4108 |
| 2 | POINT (140.11575 36.082818) | 2 | 筑波学園郵便局 | 茨城県つくば市吾妻1丁目13-2 | 0x04120000010C94C2B8 |
| 3 | POINT (140.09617 36.077463) | 3 | 葛城郵便局 | 茨城県つくば市刈間388 | 0x04120000010C35D3B8 |
| 4 | POINT (140.103989 36.063796) | 4 | 谷田部松代郵便局 | 茨城県つくば市松代4丁目200-1 | 0x04120000010CCDCEA7 |
| 5 | POINT (140.113192 36.043797) | 5 | 小野川郵便局 | 茨城県つくば市館野464 | 0x04120000010CDA54D |

➤ STDistance でつくば駅からの距離を取得

次に、STDistance 関数を利用して、TX つくば駅から 4 つの郵便局までの距離を取得してみましょう。

```

DECLARE @g1 geography
SELECT @g1 = geog FROM 郵便局 WHERE id = 1          -- TX つくば駅
SELECT @g1.STDistance(geog), * FROM 郵便局 WHERE id <> 1

```

| | (列名なし) | id | 郵便局名 | 住所 | geog |
|---|------------------|----|----------|-------------------|---------------------------|
| 1 | 377.361240496851 | 2 | 筑波学園郵便局 | 茨城県つくば市吾妻1丁目13-2 | 0x04120000010C94C2B8C7990 |
| 2 | 1505.62544631239 | 3 | 葛城郵便局 | 茨城県つくば市刈間388 | 0x04120000010C35D3BD4EEA0 |
| 3 | 2211.72329181831 | 4 | 谷田部松代郵便局 | 茨城県つくば市松代4丁目200-1 | 0x04120000010CCDCEA2772A |
| 4 | 4325.50507004059 | 5 | 小野川郵便局 | 茨城県つくば市館野464 | 0x04120000010CDA54DD239B |

筑波学園郵便局は 377 メートル、葛城郵便局は 1.5 km 離れていることを確認できます。

次に、STDistance を WHERE 句で利用して、TX つくば駅から 2km (2,000 メートル) 以内の郵便局を検索してみましょう。

```

-- TX つくば駅から 2km 以内の郵便局
DECLARE @g1 geography
SELECT @g1 = geog FROM 郵便局 WHERE id = 1
SELECT * FROM 郵便局 WHERE @g1.STDistance(geog) < 2000 AND id <> 1

```

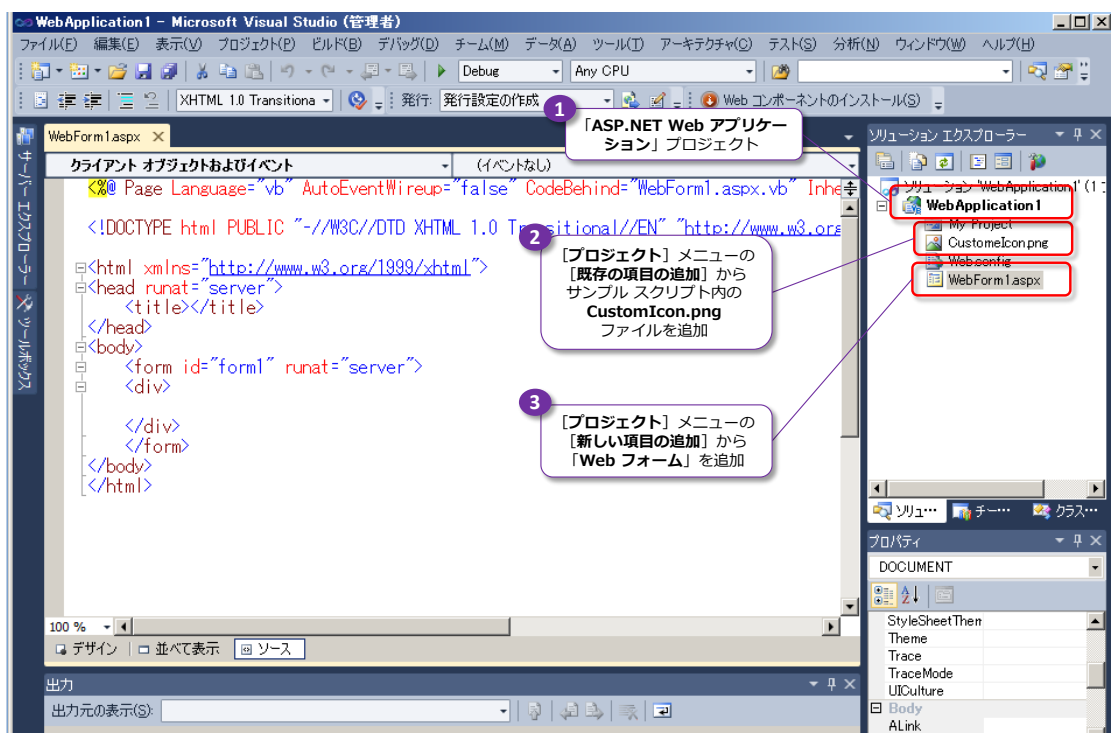
| | id | 郵便局名 | 住所 | geog |
|---|----|---------|------------------|-------------------------------------|
| 1 | 2 | 筑波学園郵便局 | 茨城県つくば市吾妻1丁目13-2 | 0x04120000010C94C2B8C7990A424062105 |
| 2 | 3 | 葛城郵便局 | 茨城県つくば市刈間388 | 0x04120000010C35D3BD4EEA094240679B1 |

筑波学園郵便局と葛城郵便局のみがヒットしていることを確認できます。

➤ Bing マップとの連携（ASP.NET Web アプリケーション）

次に、**ASP.NET 4.0** と **ADO.NET 4.0**（Visual Studio 2010、Visual Basic）を利用して、**geography** データ型のデータを取得し、それを **Bing マップ** 上へ表示する **ASP.NET Web アプリケーション**を作成してみましょう。

Bing マップへの表示部分は、「**Bing Maps AJAX Control API**」（クライアント サイドの **JavaScript**）を利用します。また、郵便局のアイコンを表示するために、サンプル スクリプト内の「**CustomIcon.png**」ファイルを利用するので、プロジェクト内へ追加しておいてください（Visual Studio 2010 の「**プロジェクト**」メニューの「**既存の項目の追加**」から、ファイルを追加することができます）。



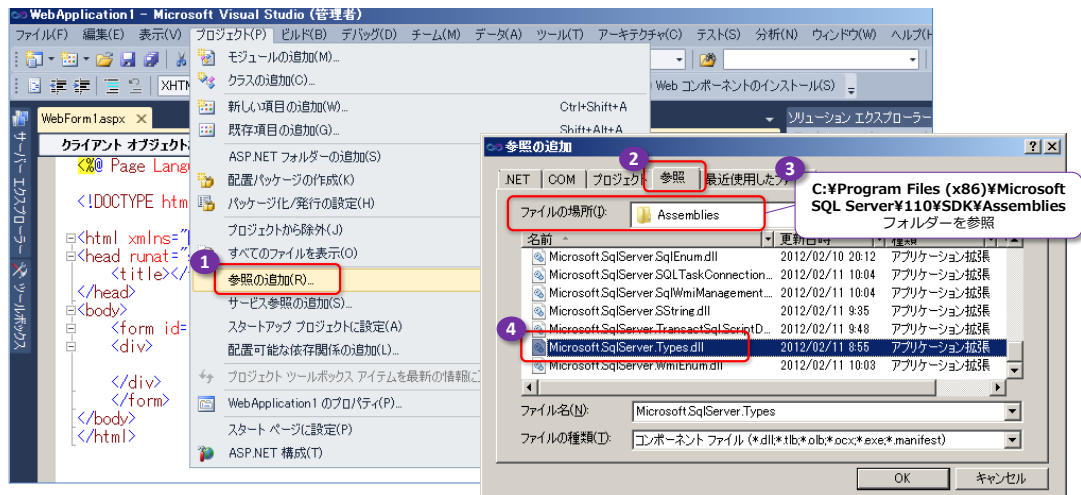
また、Visual Studio 2010 の「**プロジェクト**」メニューの「**新しい項目の追加**」から「**Web フォーム**」を選択して、新しい Web フォーム（画面では **WebForm1.aspx** ファイル）を追加しておきます。

➤ Microsoft.SqlServer.Types.dll への参照追加

geography データ型のデータを取得する部分では、**ADO.NET** を利用しますが、データを操作する部分（経度や緯度の取得など）では、**SqlGeography** オブジェクトを利用します。**SqlGeography** オブジェクトを利用するには、「**Microsoft.SqlServer.Types.dll**」ファイルへの参照を事前に追加しておく必要があります。このファイルは、次のフォルダーへ格納されています。

C:\Program Files (x86)\Microsoft SQL Server\110\SDK\Assemblies

参照の追加は、次のように「**プロジェクト**」メニューの「**参照の追加**」から行うことができます。



続いて、「**Bing Maps AJAX Cotrol API**」（クライアント サイドの **JavaScript**）を利用して、次のように単純な地図（TX つくば駅を中心にした地図）を表示するコードを記述します。



コードは、Web フォーム（**WebForm1.aspx**）のソースを次のように編集します。

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
<title>Bing Maps 連携テスト</title>

<script type="text/javascript"
src="http://ecn.dev.virtualearth.net/mapcontrol/mapcontrol.ashx?v=6.3&mkt=ja-jp"></script>
<script type="text/javascript">
    var map = null;
    function GetMap() {
        map = new VEMap('myMap');

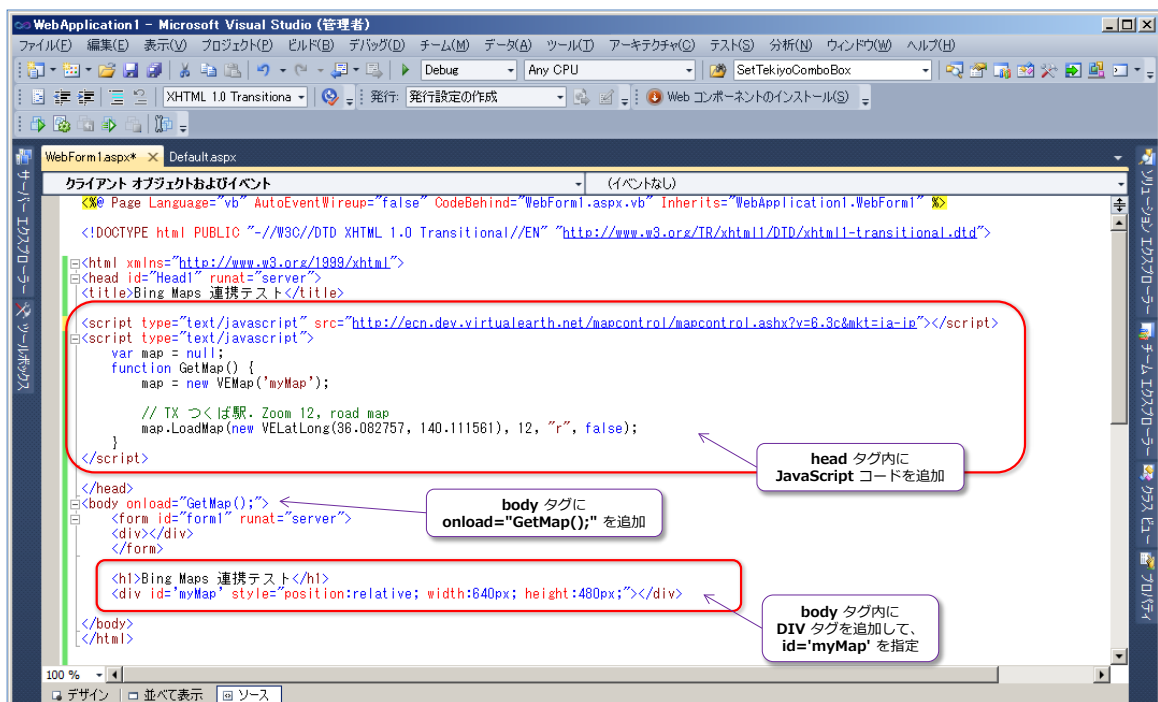
        // TX つくば駅. Zoom 12, road map
        map.LoadMap(new VELatLong(36.082757, 140.111561), 12, "r", false);
    }
</script>
</head>
<body>
</body>
</html>
```

```

</script>

</head>
<body onload="GetMap();" >
    <form id="form1" runat="server">
        <div></div>
    </form>
    <h1>Bing Maps 連携テスト</h1>
    <div id='myMap' style="position:relative; width:640px; height:480px;"></div>
</body>
</html>

```

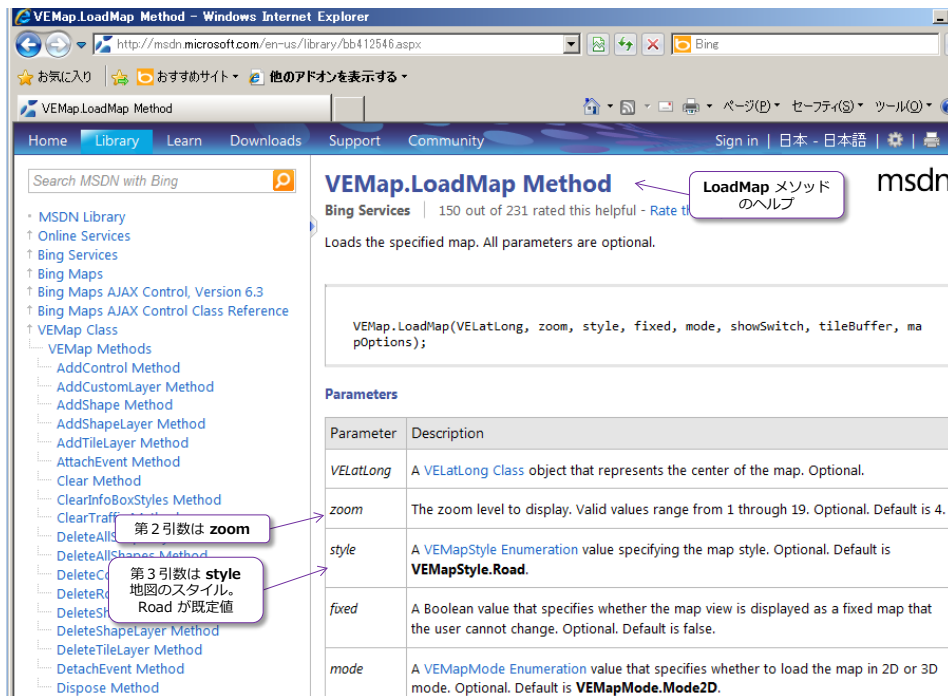


<head> タグ内の「<http://ecn.dev.virtualearth.net/mapcontrol/mapcontrol.ashx?v=6.3c>」で、利用する「Bing Maps AJAX Cotrol API」のバージョンを 6.3c へ指定し、「&mkt=ja-jp」で日本語の地図を指定します。また、<Script> タグ内で **GetMap** という名前の関数を定義し、**VEMap** オブジェクトで「**myMap**」という名前の地図（Bing Map）を作成しています。**LoadMap** メソッドで TX つくば駅の緯度と経度（36.082757, 140.111561）を指定することで、この場所を中心とした地図を表示できるようになります。第 2 引数の「12」はズーム レベル、第 3 引数の「r」は Road（道路）マップを表示するという指定です。

このように作成した地図（**myMap**）を、<body> タグ内の <div> タグで、大きさ（**width** と **Height**）を指定して、実際に表示を行っています。

こういった Bing Maps のオブジェクトやメソッドの詳細は、「**Bing Maps AJAX Cotrol, Version 6.3**」のヘルプ（リファレンス）から確認することができます。

<http://msdn.microsoft.com/en-us/library/bb412546.aspx>



➤ SqlGeography で緯度と経度を取得、VEShape でプッシュ ピンの追加

次に、ADO.NET で geography データ型のデータ（4つの郵便局の緯度と経度の生データ）を取得して、SqlGeography オブジェクトで緯度と経度を取り出す関数を作成します。取り出した緯度と経度には、Bing Maps のプッシュ ピン機能を利用して、郵便局のアイコン (CustomIcon.png ファイル) を表示するようにします。

関数は、次のように記述します (WebForm1.aspx のコード ビハインド ファイル「WebForm1.aspx.vb」へ記述します)。

```
Imports Microsoft.SqlServer.Types
Imports System.Data.SqlClient

Function pushPinStr() As String
    ' 郵便局テーブルの SELECT と SQL Server への接続文字列
    Dim sqlstr As String = "SELECT geog, 郵便局名, 住所 FROM 郵便局 WHERE id <> 1"
    Dim cnstr As String = "Server=localhost:Database=GeoTestDB;Integrated Security=SSPI"
    Dim i As Integer = 1
    Dim scriptStr As New StringBuilder("")
    Dim shapeStr As String = ""

    ' SQL Server へ接続して、郵便局テーブルを取得 (ADO.NET)
    Using cn As New SqlConnection(cnstr)
        cn.Open()
        Using cmd As New SqlCommand(sqlstr, cn)
            Using dr As SqlDataReader = cmd.ExecuteReader()
                ' 郵便局の数だけ繰り返し処理
                While dr.Read

                    ' geography データ型のデータを SqlGeography オブジェクトへ格納
                    Dim geog As New SqlGeography
                    geog = dr("geog")
                    Dim lat As String, lon As String

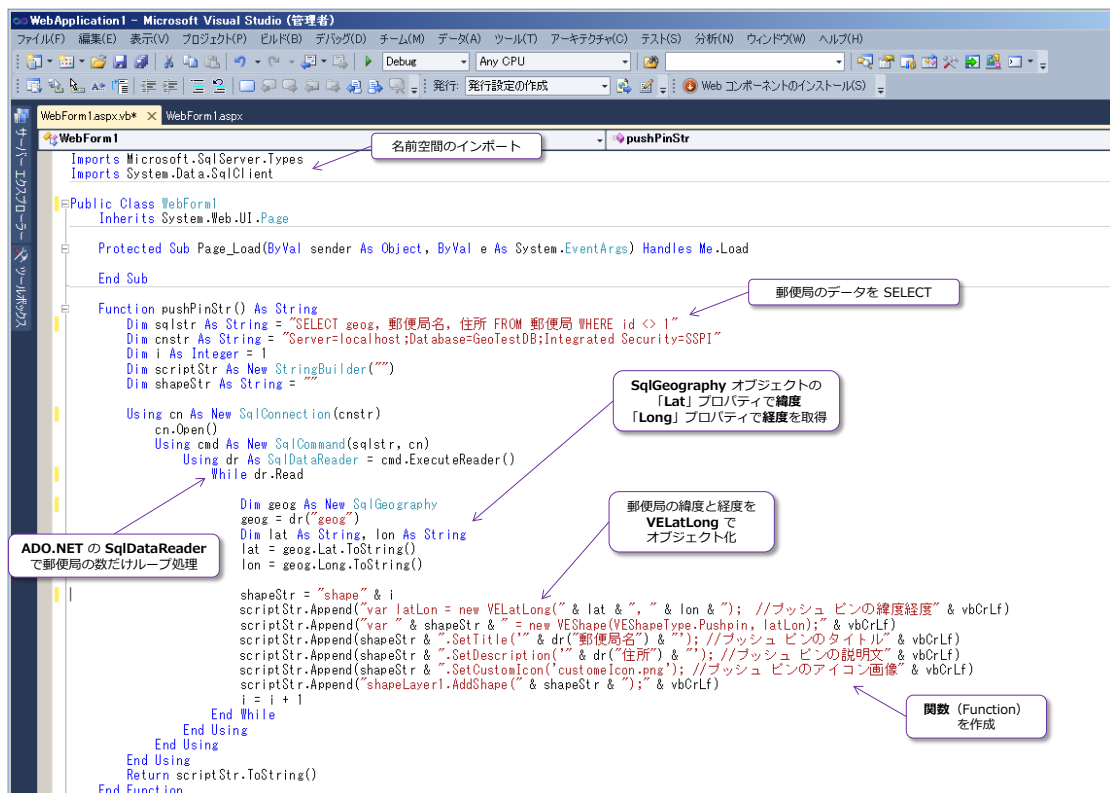
                    ' 緯度. Latitude
                    lat = geog.Lat.ToString()

                    ' 経度. Longitude
                    lon = geog.Long.ToString()
```

```

'' Bing Maps AJAX Cotrol API (JavaScript) を StringBulder で文字列として組み立て。
'' シェイプへプッシュ ピンを追加 (VELatLong と VEShape オブジェクトを利用)
'' SetCustomIcon メソッドでカスタム画像を指定
shapeStr = "shape" & i
scriptStr.Append("var latLon = new VELatLong(" & lat & ", " & lon & "); //プッシュ ピンの緯度経度" & vbCrLf)
scriptStr.Append("var " & shapeStr & " = new VEShape(VEShapeType.Pushpin, latLon);" & vbCrLf)
scriptStr.Append(shapeStr & ".SetTitle(" & dr("郵便局名") & "); //プッシュ ピンのタイトル" & vbCrLf)
scriptStr.Append(shapeStr & ".SetDescription(" & dr("住所") & "); //プッシュ ピンの説明文" & vbCrLf)
scriptStr.Append(shapeStr & ".SetCustomIcon('customIcon.png');" & vbCrLf)
scriptStr.Append("shapeLayer1.AddShape(" & shapeStr & ");" & vbCrLf)
i = i + 1
End While
End Using
End Using
Return scriptStr.ToString()
End Function

```



SqlGeography オブジェクトを利用するために「**Microsoft.SqlServer.Types**」名前空間をインポートして、このオブジェクトでは、「**Lat**」プロパティで**緯度** (Latitude)、「**Long**」プロパティで**経度** (Longitude) を取得することができます。これを Bing Maps AJAX Cotrol API の「**VELatLong**」クラスでオブジェクト化 (**latLon**) して、**VEShape** オブジェクトでプッシュ ピンとして貼り付けるようにしています (各郵便局の緯度と経度に **customIcon.gif** ファイルを表示)。

また、Web フォーム (**WebForm1.aspx**) 側のソースの **GetMap** メソッドを次のように変更します。

```

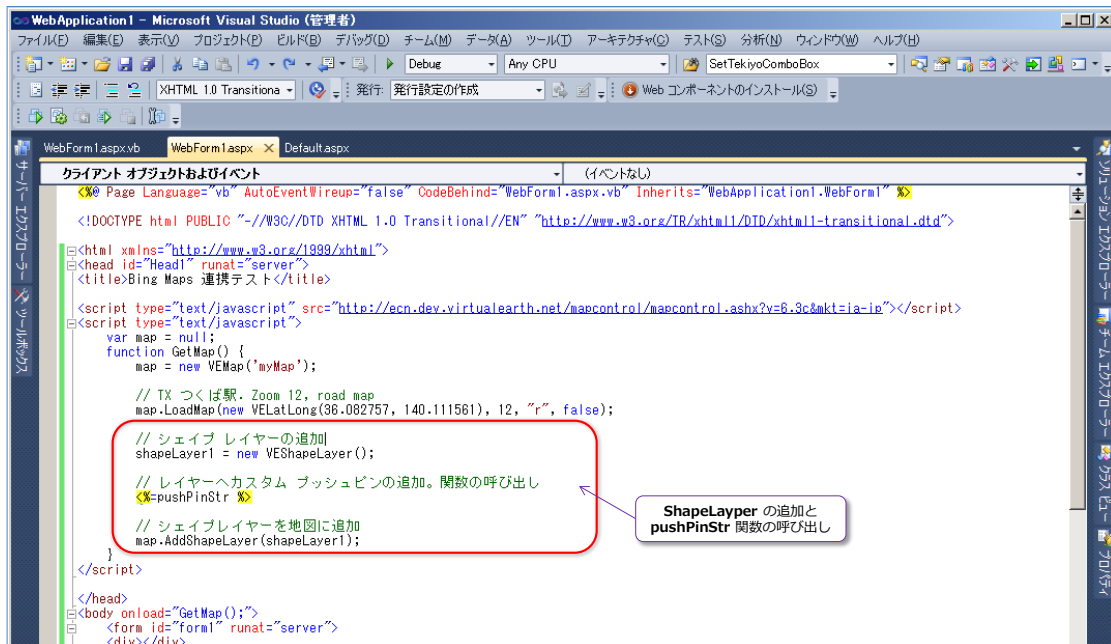
function GetMap() {
    map = new VEMap('myMap');

    // TX つくば駅. Zoom 12, road map
    map.LoadMap(new VELatLong(36.082757, 140.111561), 12, "r");
}

```



```
// シェイプ レイヤーの追加
shapeLayer1 = new VEShapeLayer();
// レイヤーへカスタム プッシュピンの追加。関数の呼び出し
<%=pushPinStr %>
// シェイプレイヤーを地図に追加
map.AddShapeLayer(shapeLayer1);
}
```



Bing Maps AJAX Control API の「**VEShapeLayer**」オブジェクトでシェイプ レイヤーを追加して、前の手順で作成した **pushPinStr** 関数の呼び出しで、プッシュ ピンのシェイプ オブジェクトを生成します。これを **AddShapeLayer** メソッドで地図へ追加することで、完成です。

デバッグ実行して、結果を確認してみましょう。



このコードは、単純に 4 つの郵便局を表示するだけですが、前述の **STDistance** 関数などと組み合わせることで、より実践的なアプリケーションを作成できるようになります。

なお、実行時に、「**SqlGeography**」でのキャスト エラーが発生する場合は、「**geog = dr("geog")**」の部分を変えて、次のように変更して実行してみてください。

```
geog = SqlGeography.Deserialize(dr.GetSqlBytes(0))
```

Note : Reporting Services の Bing Maps 連携機能

SQL Server 2012 の **Reporting Services** には、Spatial データ型 (geography/geometry) のデータをもとに、Bing Maps のレポートを作成する機能があります。これを利用すると、プログラム コードを一切記述することなく、次のようなレポートを作成することが可能です (マップ ウィザード機能によって、GUI 操作のみで作成可能です)。



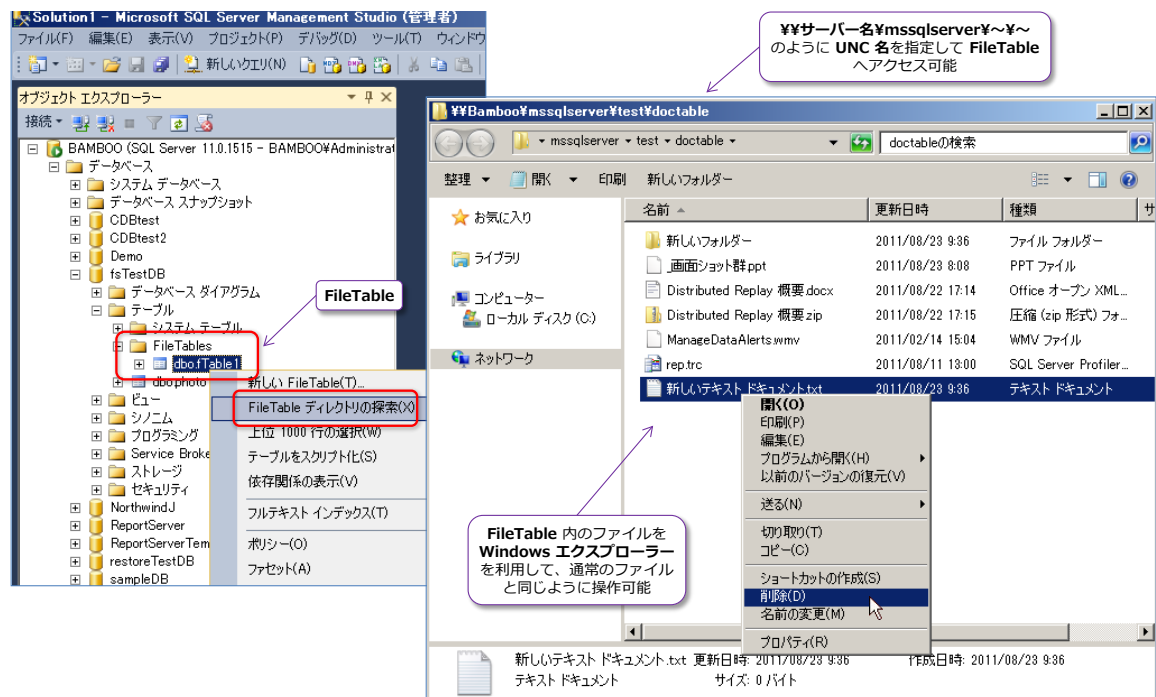
このマップ機能については、本自習書シリーズの「**Reporting Services によるレポート作成**」で詳しく説明していますので、こちらもぜひご覧いただければと思います。

5.3 FileTable による Windows ファイルのサポート

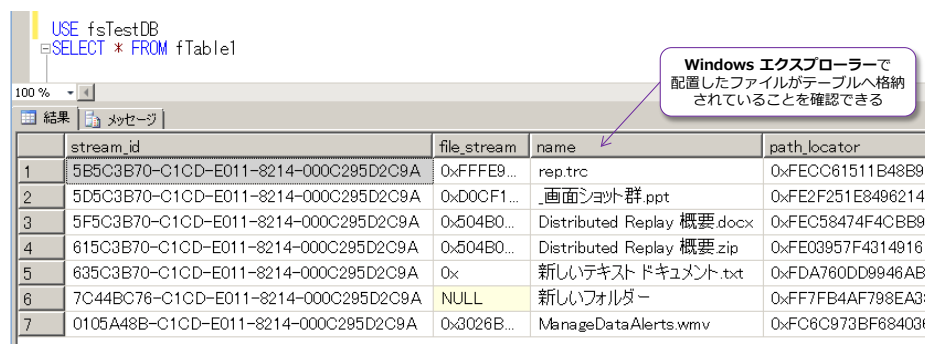
FileTable による Windows ファイルのサポート

SQL Server 2012 では、**FileTable** 機能がサポートされて、Windows 上のファイルを SQL Server 上のデータベース内に簡単に格納できるようになりました。以前のバージョンでは、**FileStream データ型**を利用して、同じことを実現できていましたが、このデータ型では、Windows ファイルを操作するために API を利用しなければなりません。これに対して、SQL Server 2012 で提供される **FileTable** 機能では、通常の Windows ファイルを操作するのと同様に Windows エクスプローラーを利用して、データベース内へファイルを格納することができるようになりました。

以下の画面は、**FileTable** に対して **Windows エクスプローラー**を利用してファイル进行操作している様子です。



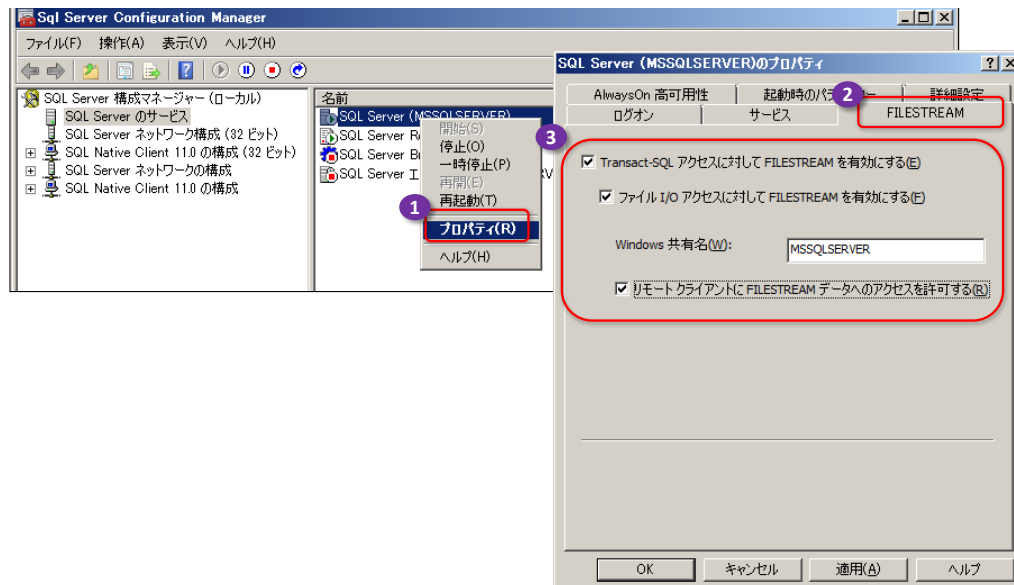
FileTable の中身を次のように **SELECT** ステートメントで参照すると、実際の Windows ファイルが SQL Server データベース内に格納されていることを確認できます。



◆ FileTable の利用方法

FileTable は、従来の **FileStream** データ型を応用した機能なので、途中までの手順が **FileStream** を利用する場合と同じです。

1. まずは **SQL Server 構成マネージャー** ツールを利用して、次のように **FileStream** 機能を有効化しておきます。



2. 次に、**sp_configure** を利用して、**filestream_access_level** 構成オプション (FileStream 機能へのアクセス レベル) を **2** へ変更します。

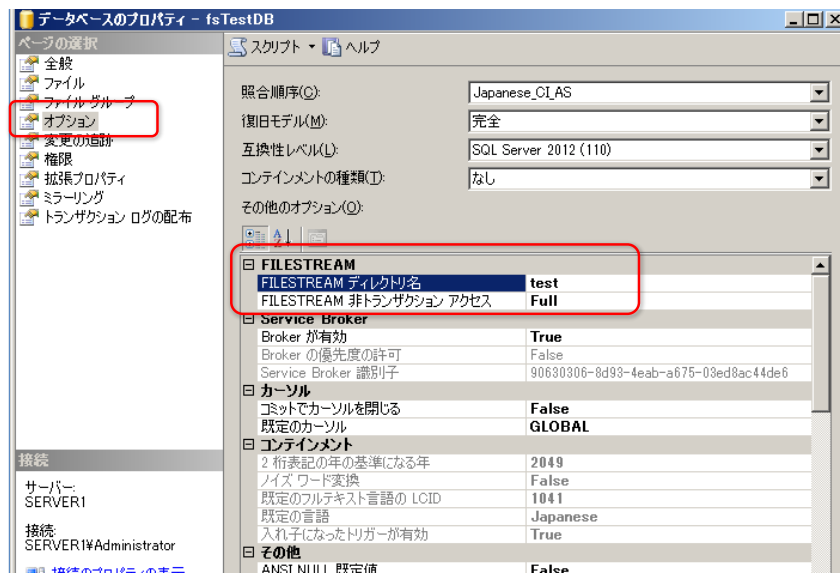
```
EXEC sp_configure 'filestream_access_level', 2
RECONFIGURE
```

3. 次に、**CREATE DATABASE** ステートメントでのデータベースの作成時に、次のように **FileStream** 用の**ファイル グループ**を作成します。

```
CREATE DATABASE fsTestDB2
ON
PRIMARY
    ( NAME = fsTestDB1_mdf
      , FILENAME = 'C:\temp\fsTestDB2.mdf' ),
FILEGROUP FileStreamGroup1 CONTAINS FILESTREAM
    ( NAME = fsTestDB_fs
      , FILENAME = 'C:\temp\fsTestDB_FileStream2' )
LOG ON
    ( NAME = fsTestDB_log
      , FILENAME = 'C:\temp\fsTestDB_Log2.ldf' )
go
```

ここまでの手順は、**FileStream** を利用する場合と全く同様です。

4. 次に、**FileTable** 機能を利用するために、[データベースのプロパティ] ダイアログを開きます。このダイアログでは、次のように [オプション] ページを開いて、[FileStream ディレクトリ名] へ任意のディレクトリ名（画面は **test**。ここで設定した名前が Windows エクスプローラーから見えるフォルダー名になります）と [FileStream 非トランザクション アクセス] を「Full」へ設定します。

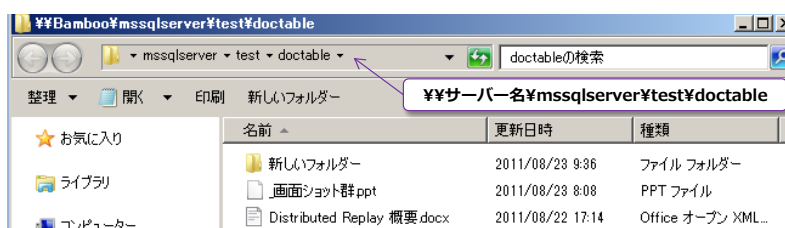


5. 次に、**CREATE TABLE** ステートメントで **FileTable** を作成します。**FileTable** を作成するには、次のように **AS FILETABLE** を指定します。

```
USE fsTestDB
CREATE TABLE fTable1 AS FILETABLE
WITH
(
    FILETABLE_DIRECTORY = 'doctable'
    , FILETABLE_COLLATE_FILENAME = database_default
)
```

FILETABLE_DIRECTORY には、任意のディレクトリ名（上記は **doctable**）を指定し、ここで設定した名前が Windows エクスプローラーから見えるフォルダー名になります。前述のデータベースのプロパティで設定したディレクトリ名（画面では **test** と設定）が上位フォルダーになり、次のパスで Windows エクスプローラーからアクセスできるようになります。

¥¥サーバー名¥mssqlserver¥test¥doctable



このように SQL Server 2012 では、**FileTable** 機能がサポートされたことで、Windows 上のファイルを SQL Server データベース内に簡単に格納できるようになりました。

➡ おわりに

最後までこの自習書の内容を試された皆さま、いかがでしたでしょうか？ Transact-SQL には、いろいろな機能が実装されていることを確認できたのではないのでしょうか。本自習書では、扱わなかったテーマとしては「リンク サーバー」や「カーソル」、「トリガー」、「XML データ型」、「SQL CLR」（CLR 統合）、「フルテキスト検索」機能などもあります。これらについては、SQL Server のオンライン ブック（Books Online）を参考に、チャレンジしてみてくださいと思います。

執筆者プロフィール

有限会社エスキューエル・クオリティ (<http://www.sqlquality.com/>)

SQLQuality (エスキューエル・クオリティ) は、**日本で唯一の SQL Server 専門の独立系コンサルティング会社**です。過去のバージョンから最新バージョンまでの SQL Server を知りつくし、多数の実績と豊富な経験を持つ、OS や .NET にも詳しい **SQL Server の専門家 (キャリア 17 年以上) がすべての案件に対応します**。人気メニューの「**パフォーマンス チューニング サービス**」は、100%の成果を上げ、過去すべてのお客様環境で驚異的な性能向上を実現。チューニング スキルは**世界トップレベル**を自負、検索エンジンでは (英語情報を含めて) ヒットしないノウハウを多数保持。ここ数年は **BI/DWH システム構築支援**のご依頼が多い。

主なコンサルティング実績

- ▶ 大手映像制作会社の **BI システム構築支援** (会計/業務システムにおける予算管理/原価管理など)
- ▶ 大手流通系の **DWH/BI システム構築支援** (POS データ/在庫データ分析)
大規模テラバイト級データ ウェアハウスの物理・論理設計支援および運用管理設計支援
- ▶ 大手アミューズメント企業の **BI システム構築支援** (人事システムにおける人材パフォーマンス管理)
- ▶ 外資系医療メーカーの Analysis Services による「**販売分析**」システムの構築支援 (売上/顧客データ分析)
- ▶ **9 TB** データベースの物理・論理設計支援 (パーティショニング対応など)
- ▶ ハードウェア リプレース時の**ハードウェア選定** (最適なサーバー、ストレージの選定)、**高可用性環境**の構築
- ▶ **SQL Server 2000** (32 ビット) から **SQL Server 2008** (x64) への移行/アップグレード支援
- ▶ 複数台の SQL Server の **Hyper-V 仮想環境**への移行支援 (サーバー統合支援)
- ▶ **2 時間**かかっていた日中バッチ実行時間を、わずか **5 分**へ短縮 (**95.8%** の性能向上)
- ▶ ピーク時の CPU 利用率 **100%** のシステムを、わずか **10%** にまで軽減し、大幅性能向上
- ▶ 平均 **185.3ms** かかっていた処理を、わずか **39.2ms** へ短縮 (**78.8%** の性能向上)
- ▶ **Java 環境** (Tomcat, Seasar2, S2Dao) の SQL Server パフォーマンス チューニング etc

コンサルティング時の作業例 (パフォーマンス チューニングの場合)

- ▶ アプリケーション コード (VB, C#, Java, ASP, VBScript, VBA) の解析/改修支援
- ▶ ストアド プロシージャ/ユーザー定義関数/トリガー (Transact-SQL) の解析/改修支援
- ▶ インデックス チューニング/SQL チューニング/ロック処理の見直し
- ▶ 現状のハードウェアで将来のアクセス増にどこまで耐えられるかを測定する高負荷テストの実施
- ▶ IIS ログの解析/アプリケーション ログ (log4net/log4j) の解析
- ▶ ボトルネック ハードウェアの発見/ボトルネック SQL の発見/ボトルネック アプリケーションの発見
- ▶ SQL Server の構成オプション/データベース設定の分析/使用状況 (CPU, メモリ, ディスク, Wait) 解析
- ▶ 定期メンテナンス支援 (インデックスの再構築/断片化解消のタイミングや断片化の事前防止策など) etc

松本美穂 (まつもと・みほ)

有限会社エスキューエル・クオリティ 代表取締役

Microsoft MVP for SQL Server (2004 年 4 月~)

経産省認定データベース スペシャリスト/MCDBA/MCSD for .NET/MCITP Database Administrator

SQL Server の日本における最初のバージョンである「SQL Server 4.21a」から SQL Server に携わり、現在、SQL Server を中心とするコンサルティングを行っている。得意分野はパフォーマンス チューニングと Reporting Services。コンサルティング業務の傍ら、講演や執筆も行い、マイクロソフト主催の最大イベント Tech・Ed などでスピーカーとしても活躍中。SE や ITPro としての経験はもちろん、記名/無記名含めて多くの執筆実績も持ち、様々な角度から SQL Server に携わってきている。著書の『SQL Server 2000 でいってみよう』と『ASP.NET でいってみよう』(いずれも翔泳社刊)は、トップ セラー (前者は 28,500 部、後者は 16,500 部発行)。近刊に『SQL Server 2012 の教科書』(ソシム刊)がある。

松本崇博 (まつもと・たかひろ)

有限会社エスキューエル・クオリティ 取締役

Microsoft MVP for SQL Server (2004 年 4 月~)

経産省認定データベース スペシャリスト/MCDBA/MCSD for .NET/MCITP Database Administrator

SQL Server の BI システムとパフォーマンス チューニングを得意とするコンサルタント。過去には、約 3,000 本のストアド プロシージャのチューニングや、テラバイト級データベースの論理・物理設計、運用管理設計、高可用性設計、BI・DWH システム設計支援などを行う。アプリケーション開発 (ASP/ASP.NET, C#, VB 6.0, Java, Access VBA など) やシステム管理者 (IT Pro) 経験もあり、SQL Server だけでなく、アプリケーションや OS、Web サーバーを絡めた、総合的なコンサルティングが行えるのが強み。Analysis Services と Excel による BI システムも得意とする。マイクロソフト認定トレーナー時代の 1998 年度には、Microsoft CPLS トレーナー アワード (Trainer of the Year) を受賞。