

Microsoft®



SQL Server 2012 自習書シリーズ No.5

Transact-SQL 入門

Published: 2008 年 4 月 6 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. Transact-SQL の概要と 自習書を試す環境について.....	5
1.1 Transact-SQL ステートメントの概要	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. Transact-SQL の構成要素.....	10
2.1 ローカル変数の利用 (DECLARE)	11
2.2 変数の宣言時の初期値代入	13
2.3 複数の変数宣言	14
2.4 バッチ (go) と変数の範囲	16
2.5 文末 (セミコロンと半角スペース)	17
2.6 コメント (-- と /* */)	18
2.7 PRINT ステートメント	19
STEP 3. 流れ制御.....	21
3.1 IF による条件分岐	22
3.2 IF ~ ELSE	24
3.3 IF EXISTS、IF NOT EXISTS	26
3.4 CASE 式による条件分岐	28
3.5 WHILE によるループ処理	30
3.6 インクリメント演算子 (+=)	31
3.7 GOTO によるジャンプ	32
3.8 WAITFOR DELAY による待機.....	33
3.9 Oracle PL/SQL との比較	34
STEP 4. 照合順序 (Collation)	35
4.1 照合順序とは	36
4.2 Japanese_CI_AS の動作 (SQL Server 2012 の既定値)	38
4.3 照合順序の種類	41
4.4 データベース単位での照合順序の設定	43
4.5 SQL ステートメント単位の照合順序の指定	46
STEP 5. データ型.....	48
5.1 データ型の種類	49
5.2 文字データ型: char、varchar	50
5.3 8,000 バイト超えの文字データ: varchar(max).....	53
5.4 Unicode データ型: nchar、nvarchar.....	54
5.5 整数型: tinyint、smallint、int、bigint.....	59
5.6 真数データ型: decimal、numeric.....	63
5.7 概数データ型: real、float	65
5.8 金額: smallmoney、money	66

5.9	日付データ型 : datetime、date、time、datetime2、datetimeoffset	68
STEP 6.	関数	74
6.1	日付と時刻に関する関数.....	75
6.2	データ型の変換関数 : CONVERT、CAST	82
6.3	文字列操作の関数	87
6.4	数値操作の関数	92
6.5	NULL 操作の関数 (ISNULL、COALESCE)	94
6.6	IIF 関数による条件分岐	95
6.7	CHOOSE 関数による指定した値の取得	96
6.8	ユーザー定義関数	97
6.9	Oracle の関数との比較.....	100

STEP 1. Transact-SQL の概要と 自習書を試す環境について

この STEP では、Transact-SQL 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。

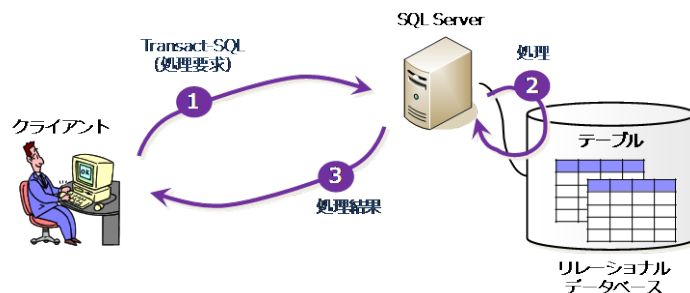
- ✓ Transact-SQL の概要
- ✓ 自習書を試す環境について
- ✓ 事前作業（sampleDB データベースの作成）

1.1 Transact-SQL ステートメントの概要

◆ Transact-SQL ステートメントの概要

Transact-SQL (T-SQL) ステートメントは、SQL Server を操作するための **“データベース操作言語”** です。テーブルに対するデータの追加や検索、更新、削除から、変数宣言や流れ制御といった **「プログラミング言語的な要素」**、バックアップや定期メンテナンスなどの **「運用管理系の操作」** まで、SQL Server に対するほとんどすべての操作（処理要求）を Transact-SQL ステートメントを利用して行うことができます。

Transact-SQL は、SQL Server を操作するための “データベース操作言語”



したがって、SQL Server を利用したアプリケーションを開発する上では、また管理者として SQL Server を管理する上でも Transact-SQL を理解しておくことが非常に重要になります。

◆ 標準 SQL (ANSI SQL92) と Transact-SQL

リレーショナル データベース (RDB) に対する基本操作となる **「SELECT」** (データの検索) や **「INSERT」** (データの追加)、**「UPDATE」** (更新)、**「DELETE」** (削除) については、**“標準 SQL”** として ANSI (米国規格協会) や ISO (国際標準化機構)、JISC (日本工業標準調査会) などの標準化団体によって規格化されています。標準 SQL の利用方法 (SQL ステートメントの基本操作) については、本自習書シリーズの **「SQL 基礎の基礎」** で詳しく説明しています。

現在、市販されているデータベース製品のほとんどは、1992 年に標準化された **「ANSI SQL92」** 規格へ準拠しています。その後、SQL 規格は、1999 年に **「SQL99」**、2003 年に **「SQL2003」**、2008 年に **「SQL2008」**、2011 年に **「SQL2011」** と規格化されていますが、これらへの準拠状況は製品によってまちまちです。

また、標準規格の SQL では、データベース サーバーを操作する上では足りない部分があるので、それを補うために各製品は独自の拡張を行っています。SQL Server の場合は、**Transact-SQL** が独自の拡張になります。そのほか、Oracle では **「PL/SQL」**、PostgreSQL では **「PL/pgSQL」** といった形で独自の拡張を行っており、これらには互換性がありません。こうした製品による SQL の違いは、**「方言」** (ダイアレクト : Dialect) とも呼ばれています。

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。

その他

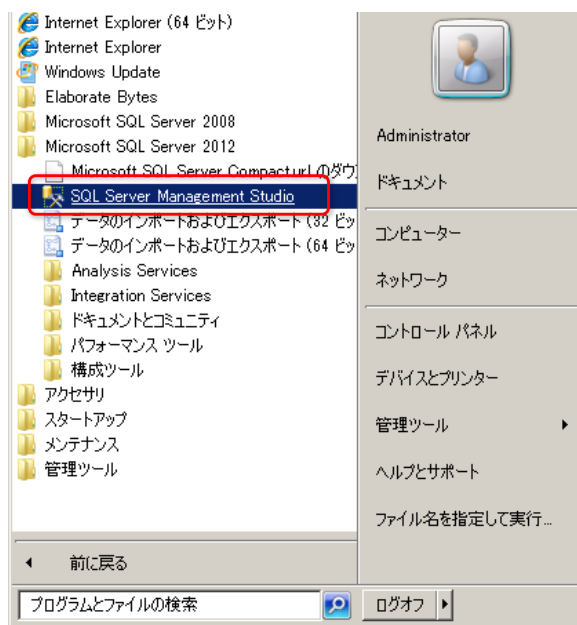
この自習書を試すには、サンプル スクリプトをダウンロードして、次のページの事前作業を実行しておく必要があります。

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）。

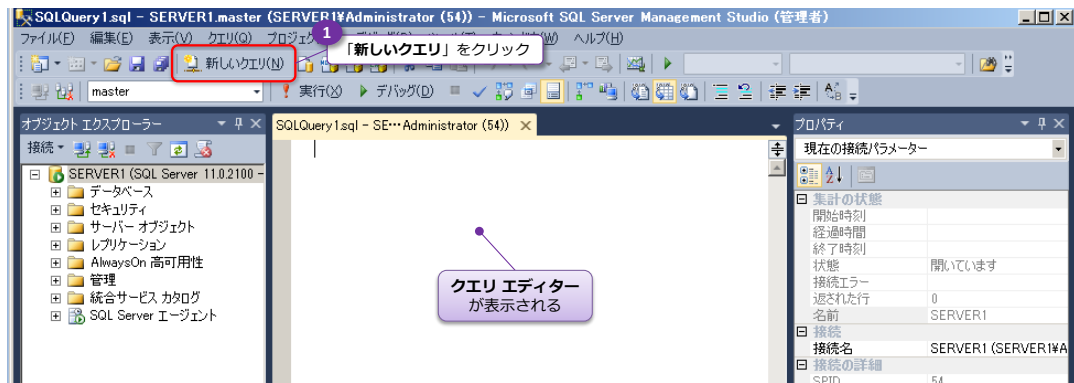
1. まずは、Management Studio を起動するために、[スタート] メニューの [すべてのプログラム] から、[Microsoft SQL Server 2012] を選択して、[SQL Server Management Studio] をクリックします。



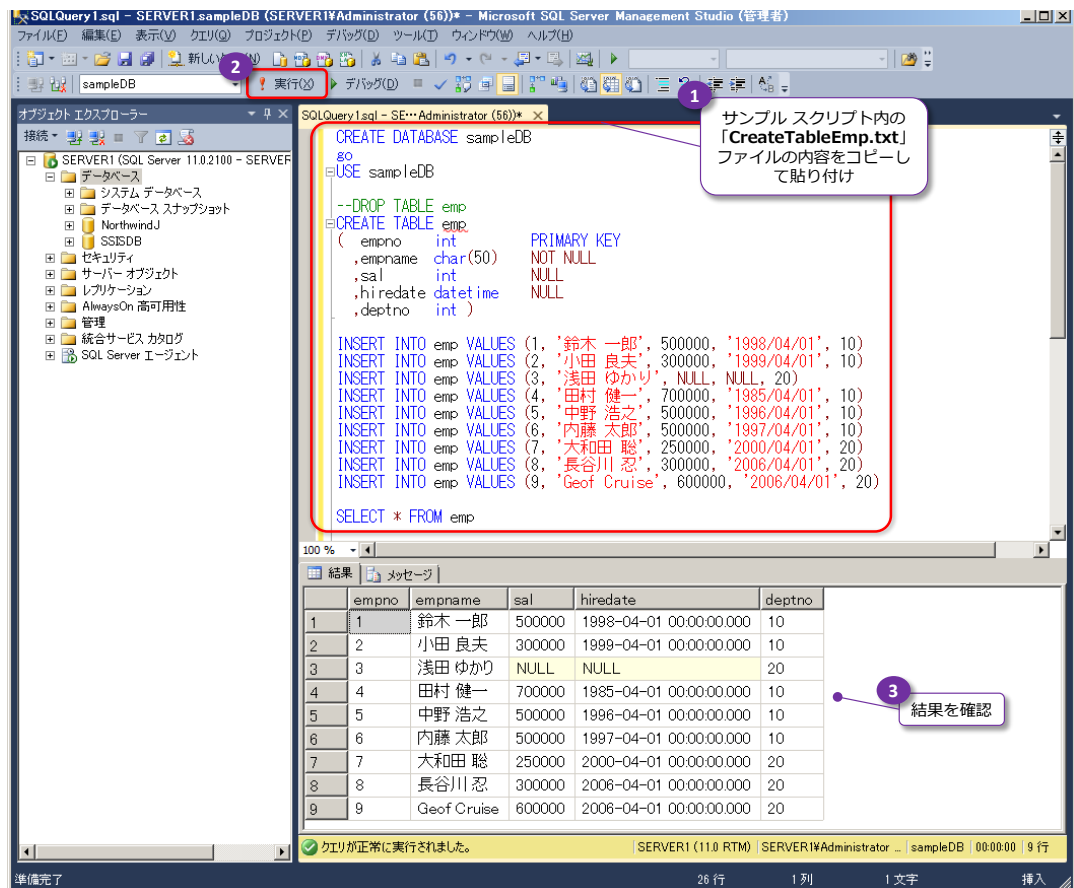
2. 起動後、次のように [サーバーへの接続] ダイアログが表示されたら、[サーバー名] へ SQL Server の名前を入力し、[接続] ボタンをクリックします。



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。このテーブルは、STEP 3 以降で利用します。

STEP 2. Transact-SQL の構成要素

この STEP では、Transact-SQL の基本の構成要素となる「**ローカル変数**」と「**バッチ**」、「**コメント**」、「**文末**」、「**PRINT ステートメント**」などを説明します。

この STEP では、次のことを学習します。

- ✓ ローカル変数の利用 (DECLARE)
- ✓ 変数宣言時の初期値代入
- ✓ 複数の変数宣言
- ✓ 変数の範囲はバッチ (go)
- ✓ 文末 (セミコロンと半角スペース)
- ✓ コメント (-- と /* */)
- ✓ PRINT ステートメント

2.1 ローカル変数の利用 (DECLARE)

ローカル変数の宣言と値の代入

Transact-SQL では、ほかのプログラミング言語と同じように変数を利用することができます。変数は、ローカル変数 (Local Variable) と呼ばれ、「**DECLARE**」ステートメントで宣言し、「**SELECT**」または「**SET**」ステートメントで値を代入します。

変数を宣言するには、DECLARE ステートメントを次のように記述します。

```
DECLARE @変数名 データ型
```

変数名の先頭には必ず「@」を付け、データ型を指定します (データ型の種類については、STEP5 で詳しく説明します)。

変数へ値を代入するには、次のように SELECT または SET ステートメントを利用します。

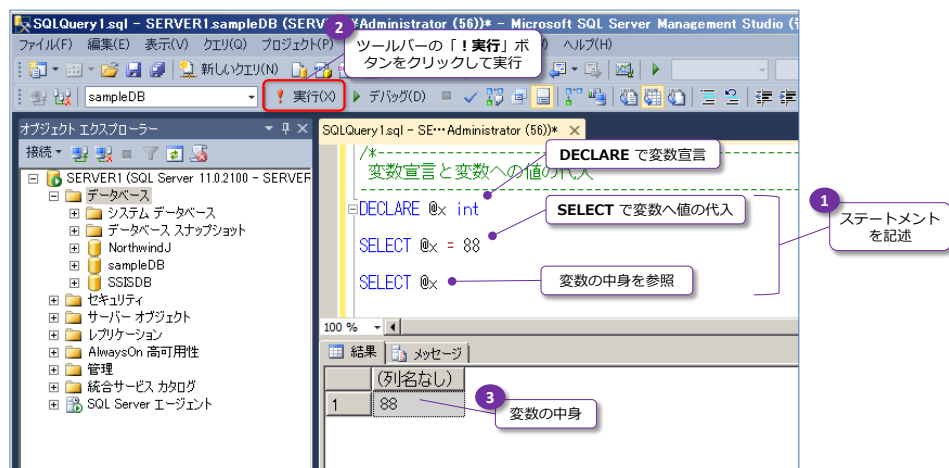
```
SELECT @変数名 = 代入したい値
または
SET @変数名 = 代入したい値
```

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**」を開いて、次のように記述して、整数を格納できるデータ型「**int**」を指定したローカル変数「**@x**」を利用してみます。

```
DECLARE @x int
SELECT @x = 88
SELECT @x
```

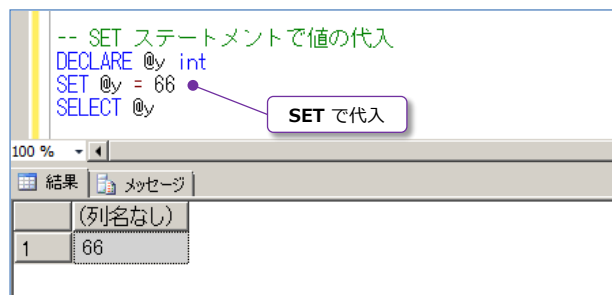


このステートメントは、DECLARE でローカル変数「@x」を宣言し、SELECT ステートメン

トで "88" という値を代入しています。最後の「**SELECT @x**」は、代入された値を確認するためのステートメントです。

- 次に、**SELECT** ステートメントの代わりに、**SET** ステートメントを利用して変数へ値を代入してみましょう。次のクエリを実行します。

```
DECLARE @y int
SET @y = 66
SELECT @y
```



このように、Transact-SQL では、変数を「**DECLARE**」ステートメントで宣言し、「**SELECT**」または「**SET**」ステートメントで値を代入します。

2.2 変数の宣言時の初期値代入

➡ 変数の宣言時の初期値代入

SQL Server 2008 からは、DECLARE ステートメントを使用して変数を宣言する際に、初期値を代入できるようになりました。これは、次のように記述します。

```
DECLARE @変数名 データ型 = 初期値
```

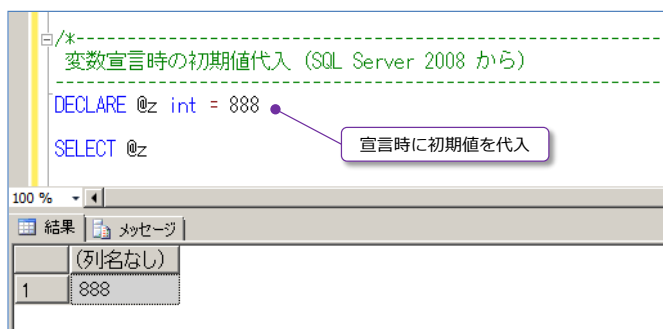
データ型に続けて「=」を記述することで、変数へ初期値を代入することができます。

➡ Let's Try

それでは、これを試してみましょう。

1. 次のように記述して、変数「@z」へ "888" という初期値を代入してみます。

```
DECLARE @z int = 888  
SELECT @z
```



このように SQL Server 2008 からは、変数の宣言時に初期値を代入できるようになったので、大変便利です。

2.3 複数の変数宣言

➡ 複数の変数宣言

DECLARE ステートメントでは、複数の変数をまとめて宣言することもできます。これは、次のように記述します。

```
DECLARE @変数1 データ型 [= 初期値], @変数2 データ型 [= 初期値], ...
```

「,」（カンマ）で区切ることで、複数の変数を宣言することができます。

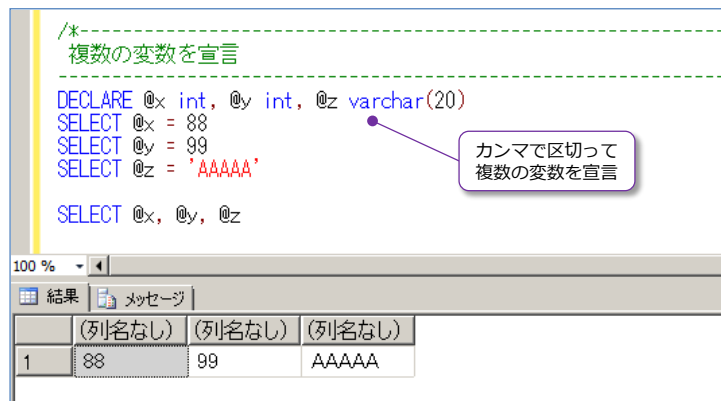
➡ Let's Try

それでは、これを試してみましょう。

1. 次のように記述して、変数「@x」と「@y」、「@z」の3つをまとめて宣言します。

```
DECLARE @x int, @y int, @z varchar(20)
SELECT @x = 88
SELECT @y = 99
SELECT @z = 'AAAAA'

SELECT @x, @y, @z
```



➡ SELECT ステートメントによる複数値の代入

2. SELECT ステートメントによる変数の代入時は、次のように複数の値をまとめて代入することも可能です。

```
DECLARE @x int, @y int, @z varchar(20)
SELECT @x = 88, @y = 99, @z = 'AAAAA'

SELECT @x, @y, @z
```

```
-- SELECT ステートメントでは、一度に代入することも可能
DECLARE @x int, @y int, @z varchar(20)
SELECT @x = 88, @y = 99, @z = 'AAAAA'

SELECT @x, @y, @z
```

SELECT ステートメントでは複数の変数に対して一度に値を代入できる

	(列名なし)	(列名なし)	(列名なし)
1	88	99	AAAAA

このように、SELECT ステートメントを利用した変数の代入では、カンマで区切って複数の変数に対して値を一度に代入できるので便利です。

なお、変数宣言時の初期値代入を利用すれば、次のようにクエリを記述することもできます。

```
DECLARE @x int = 88, @y int = 99, @z varchar(20) = 'AAAAA'
SELECT @x, @y, @z
```

```
-- 初期値代入でも可能
DECLARE @x int = 88, @y int = 99, @z varchar(20) = 'AAAAA'
SELECT @x, @y, @z
```

宣言時に複数の変数へ値を代入することもできる

	(列名なし)	(列名なし)	(列名なし)
1	88	99	AAAAA

Note : SET ステートメントでは、複数の変数を一度に扱えない

SET ステートメントを利用した変数への値の代入では、SELECT ステートメントのように複数の変数を一度に扱うことはできません（次のようにエラーが発生します）。

```
-- SET の場合はエラーとなる
DECLARE @x int, @y int, @z varchar(20)
SET @x = 88, @y = 99, @z = 'AAAAA'

SELECT @x, @y, @z
```

SET ステートメントでは、カンマで区切って、複数の変数を記述するとエラーになる

メッセージ 102、レベル 15、状態 1、行 5
,, 付近に不適切な構文があります。

したがって、SET ステートメントを利用して変数を代入する場合は、次のように 1 行ずつ（1 つの変数ごとに）行う必要があります。

```
-- SET の場合は 1 行ずつ代入しなければならない
DECLARE @x int, @y int, @z varchar(20)
SET @x = 88
SET @y = 99
SET @z = 'AAAAA'

SELECT @x, @y, @z
```

SET ステートメントでは複数の変数をいっぺんに扱うことができないので 1 行ずつ値を代入する必要がある

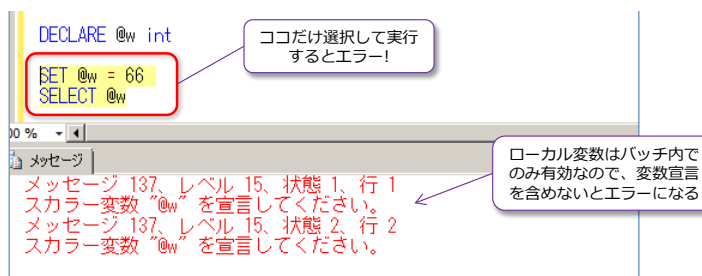
	(列名なし)	(列名なし)	(列名なし)
1	88	99	AAAAA

2.4 バッチ (go) と変数の範囲

➡ バッチと変数の範囲

Transact-SQL のローカル変数は、「**バッチ**」という範囲内でのみ有効です。バッチは、SQL Server に対してまとめて処理させたいステートメントの“塊”のことで、クエリ エディターで“**選択した範囲**”がバッチです（既定の何も選択していない状態では、クエリ エディター内へ記述したすべてのステートメントが 1 つのバッチとして扱われます）。

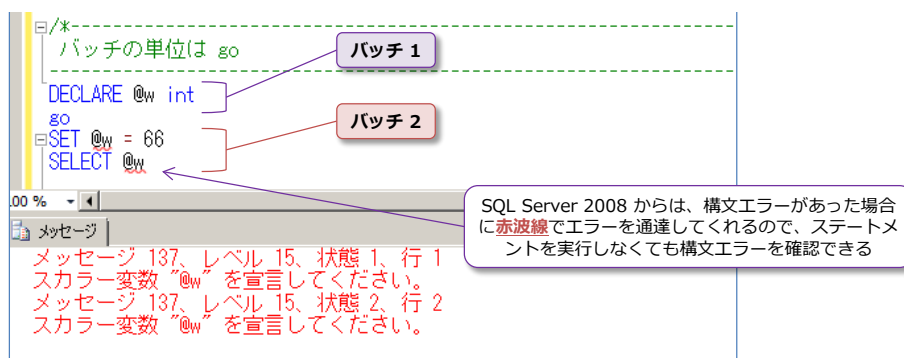
したがって、次のように DECLARE によるローカル変数の宣言を含めないで実行した場合は、エラーとなります。



➡ go コマンドによるバッチの区切り

Transact-SQL では、バッチを区切るためのコマンドとして「**go**」が用意されています（go コマンドがバッチ終了の合図になります）。

したがって、次のように DECLARE による変数宣言の後に go を記述した場合は、「変数宣言がされていない」という主旨のエラーが発生します（上述と同じエラーが発生）。



このように、ローカル変数は、バッチ内でのみ有効なので、変数宣言と変数への値の代入の間には go を入れないように注意しましょう。

2.5 文末（セミコロンと半角スペース）

➡ 文末（ステートメントの末尾・区切り）

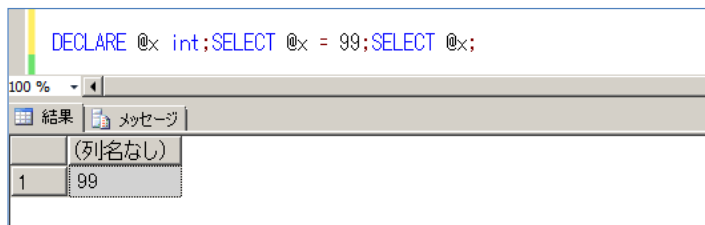
Transact-SQL では、セミコロン「;」または「**半角スペース**」があると、文末（ステートメントの末尾・区切り）と見なされます。

➡ Let's Try

それでは、これを試してみましょう。

1. 次のようにステートメントを記述して実行します。

```
DECLARE @x int;SELECT @x = 99;SELECT @x;
```

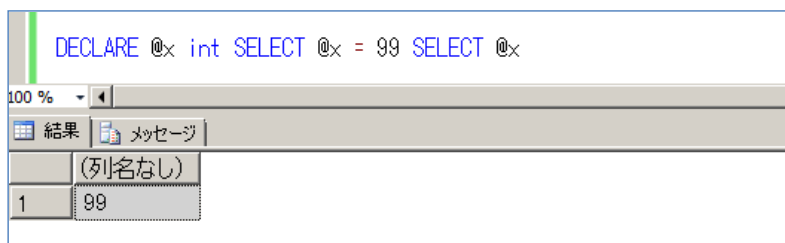


	(列名なし)
1	99

このステートメントは、「**DECLARE による変数宣言**」と「**SELECT による変数への値の代入**」、「**変数の参照**」を 1 行で記述しています。それぞれの間にセミコロン (;) を記述することで、文末（ステートメントの末尾）として扱うことができるので、このような記述が可能です。

2. 次に、セミコロンの代わりに、“**半角スペース**” を利用してみましょう。

```
DECLARE @x int SELECT @x = 99 SELECT @x
```



	(列名なし)
1	99

このように半角スペースを利用しても、文末として扱うことができます。ただ、半角スペースでステートメントを区切る場合は、一見しても少々分かりづらいので、セミコロンでステートメントを区切ることをお勧めします。

Note：半角スペースでの文末は Transact-SQL 独特

半角スペースが文末を意味するのは、Transact-SQL ならではの特徴です。多くのデータベース製品では、セミコロンのみが文末を意味します。そういった意味でも、文末は、半角スペースよりもセミコロンで区切ることをお勧めします。

2.6 コメント (-- と /* */)

◆ コメント

Transact-SQL では、「--」（ハイフン 2 つ）と、「/*」と「*/」で囲んだ範囲を“コメント”として扱うことができます。「--」は 1 行を、「/*」と「*/」は複数行をコメント化したい場合に利用します。

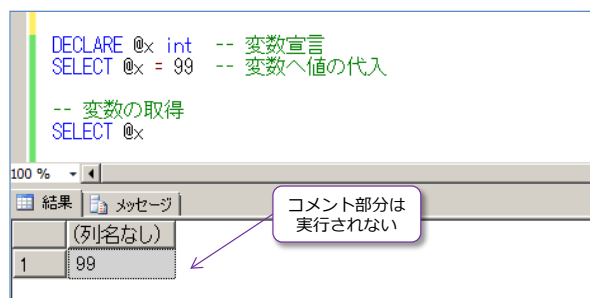
◆ Let's Try

それでは、これを試してみましょう。

1. まずは、「--」を利用して、1 行コメントを利用してみます。

```
DECLARE @x int -- 変数宣言
SELECT @x = 99 -- 変数へ値の代入

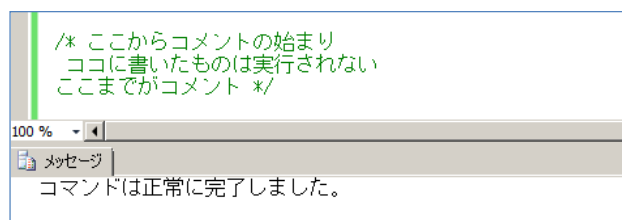
-- 変数の取得
SELECT @x
```



「--」を利用した部分はコメントとして扱われ、実行されないことを確認できます。

2. 次に、「/*」と「*/」を利用して、複数行コメントを試してみましょう。

```
/* ここからコメントの始まり
   ココに書いたものは実行されない
   ここまでがコメント */
```



このように、「/*」と「*/」で囲んだ部分は、コメントとして扱われます。

2.7 PRINT ステートメント

◆ PRINT ステートメント

PRINT ステートメントは、クエリ エディターの結果ウィンドウへメッセージを出力したい場合に利用します。これは次のように利用します。

```
PRINT '出力したい文字列'
```

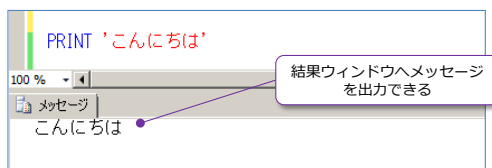
PRINT ステートメントは、アプリケーションからはほとんど利用しませんが、クエリ エディターで結果を確認したい場合や、運用管理系の SQL を実行する場合には便利なステートメントです。

◆ 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、「こんにちは」という文字列を出力してみます。

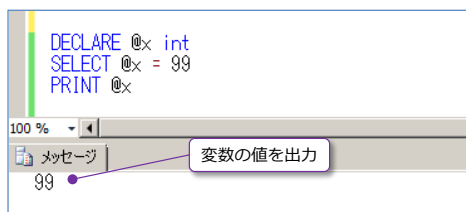
```
PRINT 'こんにちは'
```



結果ウィンドウの「メッセージ」タブへ文字列が出力されたことを確認できます。

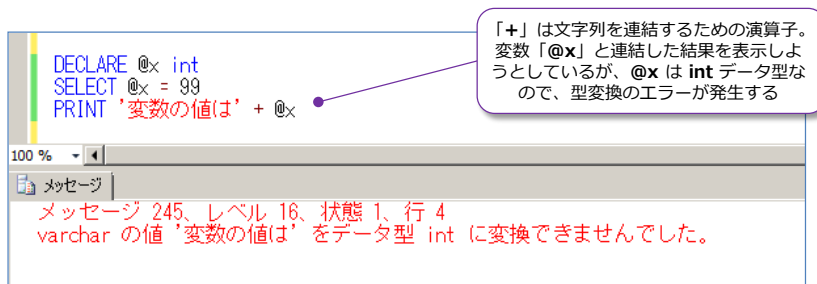
2. 次に、変数の値を PRINT ステートメントで表示してみましょう。

```
DECLARE @x int  
SELECT @x = 99  
PRINT @x
```



3. 続いて、変数の値と文字列を連結して表示してみましょう（Transact-SQL では、文字列の連結に、「+」演算子を利用します）。

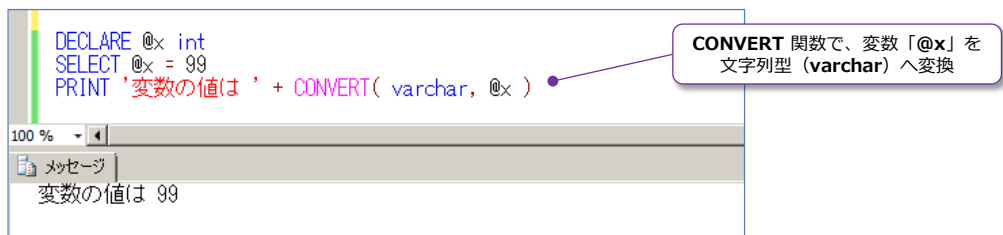
```
DECLARE @x int  
SELECT @x = 99  
PRINT '変数の値は' + @x
```



しかし、結果はエラーになります。これは変数「@x」のデータ型が **int**（整数データ）であるため、文字列との連結でデータ型が異なるという主旨のエラーです。Transact-SQL は、Visual Basic のようにデータ型があいまいな言語ではなく、型に厳しい言語なので、このようなエラーが発生します。

このエラーを回避するには、「**CONVERT**」という関数を使って、データ型を文字列型(**varchar** など)へ変換するようにします（データ型と関数については、STEP5 と STEP6 で詳しく説明します）。これは、次のように記述します。

```
DECLARE @x int
SELECT @x = 99
PRINT '変数の値は' + CONVERT( varchar, @x )
```



Note : データ型の変換 (CONVERT または CAST)

詳しくは STEP6 で説明しますが、データ型の変換には **CONVERT** または **CAST** 関数を利用します。したがって、上述の例は、次のように CAST 関数を利用しても同様の結果を得られます。

```
DECLARE @x int
SELECT @x = 99
PRINT '変数の値は' + CAST( @x AS varchar)
```

メッセージ
変数の値は 99

STEP 3. 流れ制御

この STEP では、Transact-SQL で利用できる流れ制御構文について説明します。条件分岐が行える「**IF**」や「**IF EXISTS**」、「**CASE 式**」、ループ処理の「**WHILE**」などを説明します。いずれもよく利用する基本の流れ制御構文になるので、確実にマスターしておきましょう。

この STEP では、次のことを学習します。

- ✓ IF による条件分岐
- ✓ IF ~ ELSE
- ✓ IF EXISTS、IF NOT EXISTS による存在チェック
- ✓ CASE 式による条件分岐
- ✓ WHILE によるループ処理
- ✓ インクリメント演算子（+=）
- ✓ GOTO によるジャンプ
- ✓ WAITFOR DELAY による待機

3.1 IF による条件分岐

◆ IF キーワードによる条件分岐

Transact-SQL では、**IF** キーワードを利用することで、条件によって処理を分岐することができます。構文は、次のとおりです。

```
IF (条件式)
[ BEGIN ]
    条件が真 (true) の場合に実行したいステートメント
[ END ]
```

条件式のカッコ () は、省略することができます。また、BEGIN と END は、実行したいステートメントが 1 つの場合には、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、次のようにステートメントを記述します。

```
DECLARE @x int
SET @x = DATEPART ( hour, GETDATE() )

IF @x < 12
BEGIN
    PRINT 'おはよう'
END
```

GETDATE 関数で現在時刻を取得

DATEPART 関数で現在時刻のうち、時間 (hour) のみを取得

このステートメントは、**GETDATE** という関数で現在時刻を取得し、**DATEPART** という関数で現在の時刻のうちの時間のみを取得し、それを変数「@x」へ格納しています（関数については STEP6 で説明します）。

IF キーワードでは、条件式を「@x < 12」と記述することで、現在時刻が 12 時より前かどうかで条件分岐ができ、12 時より前なら「おはよう」と表示され、そうでない場合には「**コマンドは正常に完了しました**」と表示されます。

```
DECLARE @x int
SET @x = DATEPART ( hour, GETDATE() )

IF @x < 12
BEGIN
    PRINT 'おはよう'
END
```

0時～12時前



メッセージ
おはよう



それ以外

メッセージ
コマンドは正常に完了しました。

➡ 条件式にカッコを付ける

IF キーワードで指定する条件式は、次のようにカッコを付けても同じ意味になります。

```
IF ( @x < 12 )  
  BEGIN  
    PRINT 'おはよう'  
  END
```

➡ BEGIN ～ END の省略

IF の中で実行したいステートメントが 1 つだけの場合は、BEGIN と END を省略して、次のように記述することもできます。

```
IF @x < 12  
  PRINT 'おはよう'
```

Note : BEGIN ～ END の省略しすぎに注意

BEGIN と END の省略は、あくまでも実行したいステートメントが 1 つの場合のみであることに注意してください。たとえば、次のように記述したとします。

```
IF @x < 12  
  PRINT 'おはよう'  
  SELECT * FROM emp
```

これは、IF の中で「SELECT * FROM emp」を実行しようとしています。実際には次のように解釈されます。

```
IF @x < 12  
  BEGIN  
    PRINT 'おはよう'  
  END  
  SELECT * FROM emp
```

これでは、条件に関係なく「SELECT * FROM emp」が実行されてしまいます。したがって、BEGIN と END の省略には、十二分に注意するようにしましょう。

3.2 IF ～ ELSE

➡ IF ～ ELSE

Transact-SQL では、**ELSE** キーワードを利用することで、IF の条件が満たされなかった場合の動作を記述できるようになります。構文は、次のとおりです。

```
IF 条件式
[ BEGIN ]
    条件が真 (true) の場合に実行したいステートメント
[ END ]
ELSE
[ BEGIN ]
    条件が偽 (false) の場合に実行したいステートメント
[ END ]
```

➡ Let's Try

それでは、これを試してみましょう。

1. 前の STEP で利用したステートメントに対して、次のように ELSE キーワードを追加して、条件が満たされなかった場合の処理を記述します。

```
DECLARE @x int
SET @x = DATEPART( hour, GETDATE() )

IF @x < 12
BEGIN
    PRINT 'おはよう'
END
ELSE
BEGIN
    PRINT 'こんにちは'
ENDIF
```

```
DECLARE @x int
SET @x = DATEPART( hour, GETDATE() )

IF @x < 12
BEGIN
    PRINT 'おはよう'
END
ELSE
BEGIN
    PRINT 'こんにちは'
END
```

0時～12時前

メッセージ
おはよう

それ以外

メッセージ
こんにちは

結果は、現在時刻が朝の 0 時～昼の 12 時より前なら「おはよう」と表示され、それ以外なら「こんにちは」と表示されます。

IF と ELSE で実行するステートメントは、それぞれ 1 つだけなので、BEGIN と END を省略して次のように記述することも可能です。

```
IF @x < 12
    PRINT 'おはよう'
ELSE
    PRINT 'こんにちは'
```

➡ IF の入れ子

次に、IF を入れ子にして（IF の中に IF を入れて）実行してみましょう。今まで試したステートメントと同じように現在時刻を使って、次のようにメッセージが表示されるようにします。

- ・朝 0 時～昼 12 時までは「**おはよう**」
- ・昼 12 時～夕方 5 時（17 時）までは「**こんにちは**」
- ・夕方 5 時（17 時）以降は「**こんばんは**」

```
DECLARE @x int
SET @x = DATEPART( hour, GETDATE() )

IF @x < 12
    BEGIN
        PRINT 'おはよう'
    END
ELSE
    BEGIN
        IF @x < 17
            BEGIN
                PRINT 'こんにちは'
            END
        ELSE
            BEGIN
                PRINT 'こんばんは'
            END
    END
END
```

このように複数の条件がある場合は、IF を入れ子（ネスト）にして利用します（Transact-SQL には、他の言語にあるような ElseIf が存在しないので、IF を入れ子にしなければなりません）。

3.3 IF EXISTS、IF NOT EXISTS

➡ IF EXISTS、IF NOT EXISTS

IF EXISTS または **IF NOT EXISTS** を利用すると、SELECT ステートメントによる検索結果があるか、ないかで条件分岐をできるようになります。Exist は「存在する」という意味です。

構文は、次のとおりです。

```
IF [ NOT ] EXISTS ( SELECT ステートメント )
[ BEGIN ]
    データがある場合に実行したいステートメント
[ END ]
ELSE
[ BEGIN ]
    データがない場合に実行したいステートメント
[ END ]
```

IF EXISTS のカッコに SELECT ステートメントを記述し、そのステートメントの結果がある場合 (Exist の場合) は、BEGIN と END 内のステートメントが実行されます。IF NOT EXISTS は、逆の順番で処理したい場合に利用します。

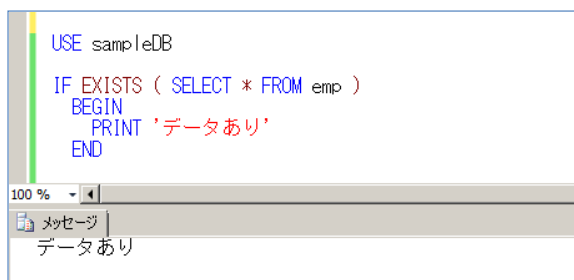
➡ Let's Try

それでは、これを試してみましょう。

1. まずは、「sampleDB」データベースの「emp」テーブルを SELECT するクエリを IF EXISTS へ指定してみます。

```
USE sampleDB

IF EXISTS ( SELECT * FROM emp )
BEGIN
    PRINT 'データあり'
END
```

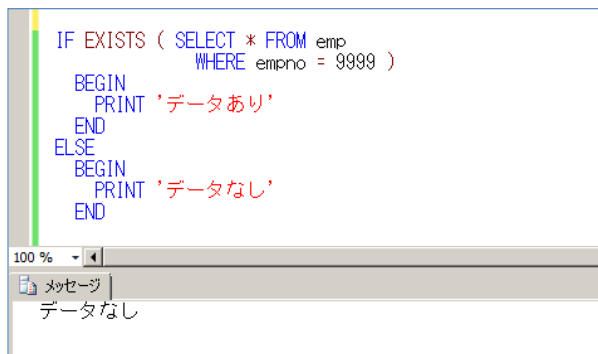


emp テーブル内にはデータが 9 件格納されているので (データが存在するので)、PRINT ス

ステートメントが実行されます。

- 次に、WHERE 句の条件として「**empno=9999**」を追加して、ELSE キーワードも追加して実行してみましょう。

```
IF EXISTS ( SELECT * FROM emp
             WHERE empno = 9999 )
BEGIN
    PRINT 'データあり'
END
ELSE
BEGIN
    PRINT 'データなし'
END
```



emp テーブルには、empno が「9999」の社員は存在しないので、ELSE 内の PRINT ステートメントが実行されることを確認できます。

なお、データが存在しない場合にだけ処理を実行したい場合は、次のように IF NOT EXISTS を利用すると便利です。

```
IF NOT EXISTS ( SELECT * FROM emp
                 WHERE empno = 9999 )
BEGIN
    PRINT 'データなし'
END
```

3.4 CASE 式による条件分岐

➡ CASE 式による条件分岐

CASE 式は、Visual Basic での「Select Case」、C 言語や Java での「switch」と同じように、複数の条件分岐が行える非常に便利な式です。構文は、次のとおりです。

```
CASE [ 式 ]
  WHEN 条件 1 THEN 条件 1 を満たした場合の値
  WHEN 条件 2 THEN 条件 2 を満たした場合の値
  :
  ELSE すべての条件を満たしていない場合の値
END
```

CASE 式は、IF や IF EXISTS のように単独で利用することはできませんが、SELECT ステートメント内など、ステートメント内の一部として利用することができます。

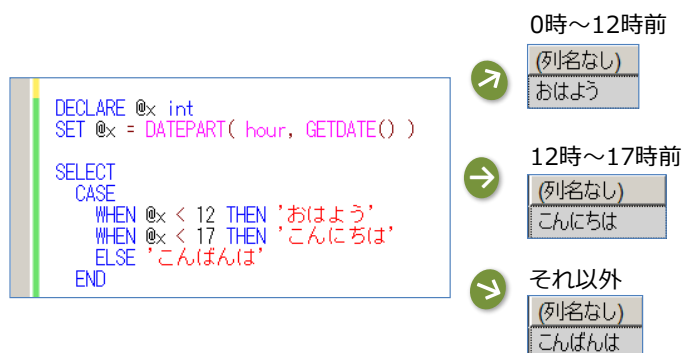
➡ Let's Try

それでは、これを試してみましょう。

1. まずは、前の STEP の IF の入れ子で試したのと同様の複数の条件分岐を行う CASE 式を試してみましょう。

```
DECLARE @x int
SET @x = DATEPART( hour, GETDATE() )

SELECT
CASE
  WHEN @x < 12 THEN 'おはよう'
  WHEN @x < 17 THEN 'こんにちは'
  ELSE 'こんばんは'
END
```



結果は、現在時刻が朝の 0 時～昼の 12 時より前なら「おはよう」、昼の 12 時から 17 時前なら「こんにちは」、それ以外なら「こんばんは」と表示されます。

➡ CASE 式の注意点

前述したように CASE 式は、SELECT ステートメント内など、ステートメント内の一部として利用することはできますが、IF や IF EXISTS のように “単独” で利用することはできません。したがって、次のように CASE 式の THEN へ別個のステートメント (PRINT ステートメントなど) を記述した場合は構文エラーになります。

The screenshot shows a SQL query in a query editor:

```
DECLARE @x int
SET @x = DATEPART( hour, GETDATE() )

SELECT
CASE
WHEN @x < 12 THEN PRINT 'おはよう'
WHEN @x < 17 THEN PRINT 'こんにちは'
ELSE PRINT 'こんばんは'
END
```

A callout box points to the PRINT statements with the text: "CASE 式内へ別個のステートメントを記述すると構文エラーとなる" (Describing a separate statement inside a CASE expression causes a syntax error).

The message window at the bottom shows two error messages:

- メッセージ 156、レベル 15、状態 1、行 6
キーワード 'PRINT' 付近に不適切な構文があります。
- メッセージ 102、レベル 15、状態 1、行 9
'END' 付近に不適切な構文があります。

➡ 変数への代入時に CASE 式を利用

CASE 式は、変数への値の代入時に利用すると便利です。前述の例とほとんど同じですが、次のように利用することができます。

The screenshot shows a SQL query in a query editor:

```
DECLARE @x int, @msg varchar(20)
SET @x = DATEPART( hour, GETDATE() )

SELECT @msg =
CASE
WHEN @x < 12 THEN 'おはよう'
WHEN @x < 17 THEN 'こんにちは'
ELSE 'こんばんは'
END

PRINT @msg
```

Three message windows are shown, each preceded by a green arrow icon, indicating the output of the PRINT statement:

- 0時～12時前
メッセージ
おはよう
- 12時～17時前
メッセージ
こんにちは
- それ以外
メッセージ
こんばんは

「SELECT @msg='おはよう'」や「SELECT @msg='こんにちは'」など、@msg に代入する値を @x の値に応じて、変更しています。前の STEP で IF の入れ子で記述した例を、CASE 式を利用することによってシンプルに記述できるようになります。

3.5 WHILE によるループ処理

➡ WHILE によるループ処理

WHILE は、繰り返し処理（ループ）を行わせたい場合に利用します。構文は、次のとおりです。

```
WHILE (条件式)
[ BEGIN ]
    条件を満たしている間、実行したいステートメント
[ END ]
```

IF キーワードのときと同様、条件式のカッコは省略可能です。また、実行したいステートメントが 1 つの場合には、BEGIN と END を省略することができます。

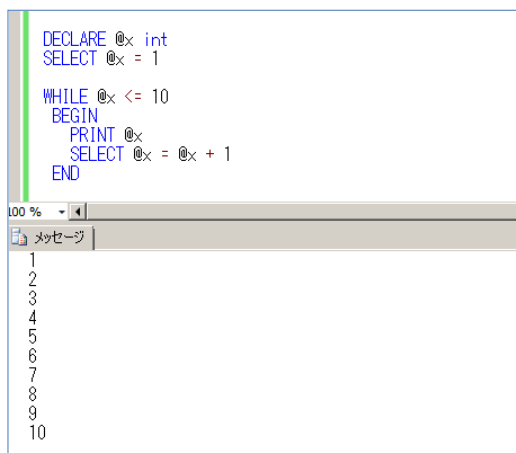
➡ Let's Try

それでは、これを試してみましょう。

1. 次のように、変数「@x」に対して「@x <= 10」という条件式を設定して、1 から 10 までの間ループするようにします。

```
DECLARE @x int
SELECT @x = 1

WHILE @x <= 10
BEGIN
    PRINT @x
    SELECT @x = @x + 1
END
```



```
DECLARE @x int
SELECT @x = 1

WHILE @x <= 10
BEGIN
    PRINT @x
    SELECT @x = @x + 1
END
```

100 %

メッセージ

1
2
3
4
5
6
7
8
9
10

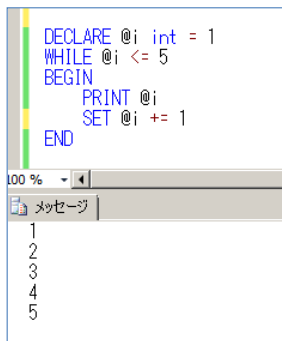
このように WHILE を利用すると繰り返し処理が行えるようになるので、パフォーマンス検証の際のテスト データの生成などに利用すると便利です。

3.6 インクリメント演算子（+=）

➡ インクリメント演算子

SQL Server 2008 からは、「+=」を利用したインクリメント演算子がサポートされました。これは、次のように利用することができます。

```
DECLARE @i int = 1
WHILE @i <= 5
BEGIN
    PRINT @i
    SET @i += 1
END
```

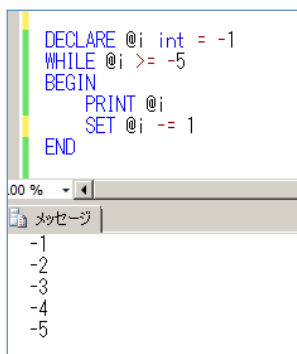


この「SET @i += 1」という記述は、「SET @i = @i + 1」と等価です（SET の代わりに SELECT と用いても同等です）。

➡ デクリメント演算子（-=）

デクリメント演算子としては、「-=」がサポートされたので、次のように利用することができます。

```
DECLARE @i int = -1
WHILE @i >= -5
BEGIN
    PRINT @i
    SET @i -= 1    -- SET @i = @i - 1 と等価
END
```



3.7 GOTO によるジャンプ

➡ GOTO によるジャンプ

GOTO ステートメントは、あまり利用しませんが、任意の場所（指定したラベル）へジャンプできるようにになります。構文は、次のとおりです。

```
GOTO ラベル
...
ラベル :
    実行したいステートメント
```

➡ Let's Try

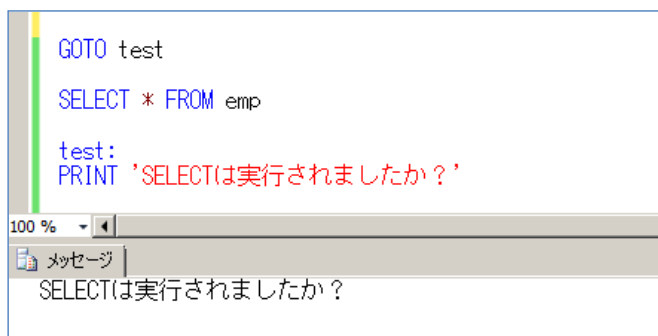
それでは、これを試してみましょう。

1. 次のように「**test**」というラベルを記述して、GOTO ステートメントでジャンプさせてみましょう。

```
GOTO test

SELECT * FROM emp

test:
PRINT 'SELECTは実行されましたか？'
```



GOTO ステートメントによって、間の SELECT ステートメントが実行されなかったことを確認できます。

3.8 WAITFOR DELAY による待機

➡ WAITFOR DELAY による待機

WAITFOR DELAY ステートメントは、処理を数秒間待機したい場合に利用します。Wait は「待つ」、Delay は「遅延」という意味です。構文は、次のとおりです。

```
WAITFOR DELAY '待機したい時間'
```

➡ Let's Try

それでは、これを試してみましょう。

1. 次のように記述して、3 秒待機した後に、SELECT ステートメントを実行してみます。

```
WAITFOR DELAY '00:00:03'  
  
SELECT * FROM emp
```

実行後すぐには SELECT ステートメントは実行されず、3 秒間待った後に実行されることを確認できます。

このように WAITFOR DELAY は、処理を数秒間待機したい場合に利用します。動作検証を行う場合などで意外と役立つ便利なステートメントです。

3.9 Oracle PL/SQL との比較

➡ Oracle PL/SQL との比較

Oracle では、Transact-SQL のようなプログラミング言語要素は、「**PL/SQL**」として実装されています。PL/SQL と Transact-SQL を比較すると次のようになります。

	Oracle (PL/SQL)	SQL Server (Transact-SQL)
基本形	DECLARE 変数宣言部 BEGIN 実行部 (必須) EXCEPTION 例外処理部 END;	<ul style="list-style-type: none"> ・フリーフォーマット ・ ; は任意 ・どこでも変数宣言可能 ・例外処理は TRY ~ CATCH (応用編で解説)
変数宣言と値の代入	DECLARE 変数名1 データ型 := 初期値; 変数名2 データ型 := 初期値; BEGIN 変数名1 := 値; 変数名2 := 値; : END;	DECLARE @変数名 データ型 = 初期値 SELECT @変数名 = 値 または SET @変数名 = 値
コメント	1行コメント -- 複数行コメント /* */	同じ
文字列出力	DBMS_OUTPUT.PUT_LINE('文字列')	PRINT '文字列'
IF による条件分岐	IF 条件式 THEN ステートメント; ELSIF 条件式 THEN ステートメント; ELSE ステートメント; END IF;	IF 条件式 [BEGIN] ステートメント [END] ELSE [BEGIN] ステートメント [END]
WHILE ループ	WHILE 条件式 LOOP ステートメント; END LOOP;	WHILE 条件式 [BEGIN] ステートメント [END]
LOOP	LOOP ステートメント; EXIT WHEN 条件; ステートメント; END LOOP;	なし
GOTO	GOTO LABEL; : <<LABEL>>	GOTO LABEL : LABEL:

STEP 4. 照合順序 (Collation)

この STEP では、SQL Server で文字列を検索（照合・比較）する際の "英字の大文字と小文字を区別するかどうか" や "半角と全角を区別するかどうか" などを決定する機能となる「**照合順序**」について説明します。

この STEP では、次のことを学習します。

- ✓ 照合順序とは
- ✓ Japanese_CI_AS の動作 (SQL Server 2012 の既定値)
- ✓ 照合順序の種類
- ✓ データベース単位での照合順序の設定
- ✓ SQL ステートメント単位の照合順序の指定

4.1 照合順序とは

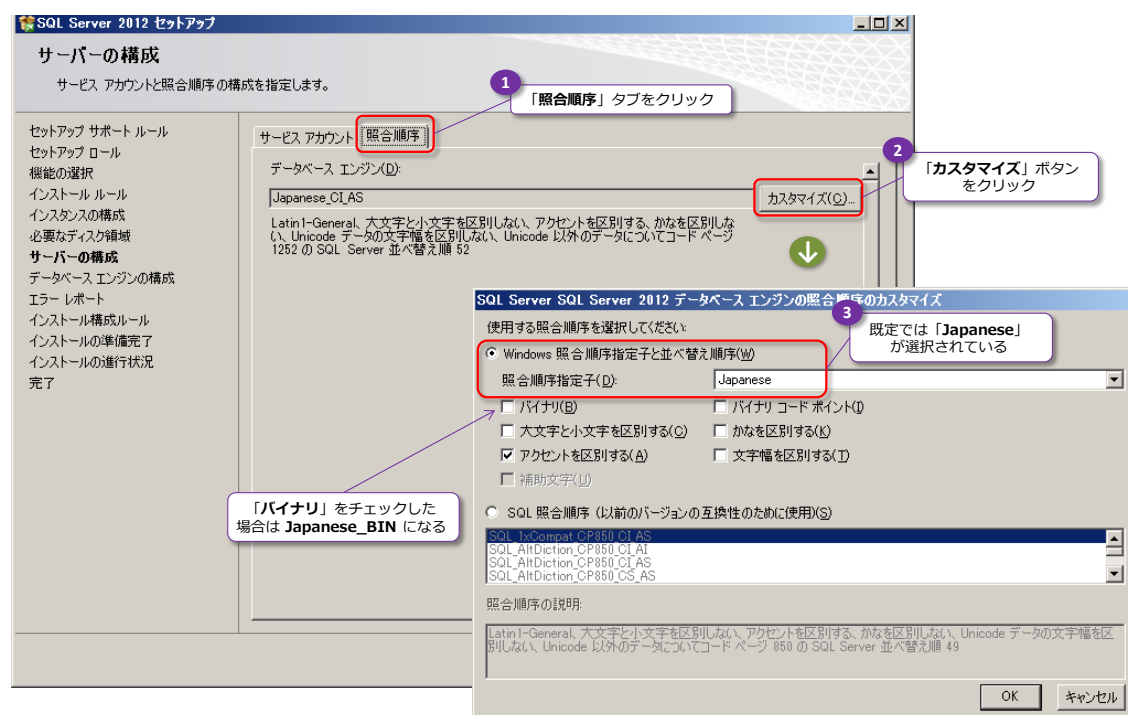
◆ 照合順序とは

照合順序は、SQL Server で文字列を検索する際の "英字の大文字と小文字を区別するか" や "カタカナとひらがなを区別するか"、"半角と全角を区別するか" などの、文字列の照合（比較）を決定するための機能です。

照合順序には、「SQL Server レベルでの設定」と「データベース単位での設定」、「テーブルの列単位での設定」の 3 種類がありますが、SQL Server レベルでの設定は、SQL Server のインストール時に設定し、その設定は再セットアップをしない限りは変更することができません。

◆ インストール時の照合順序の設定

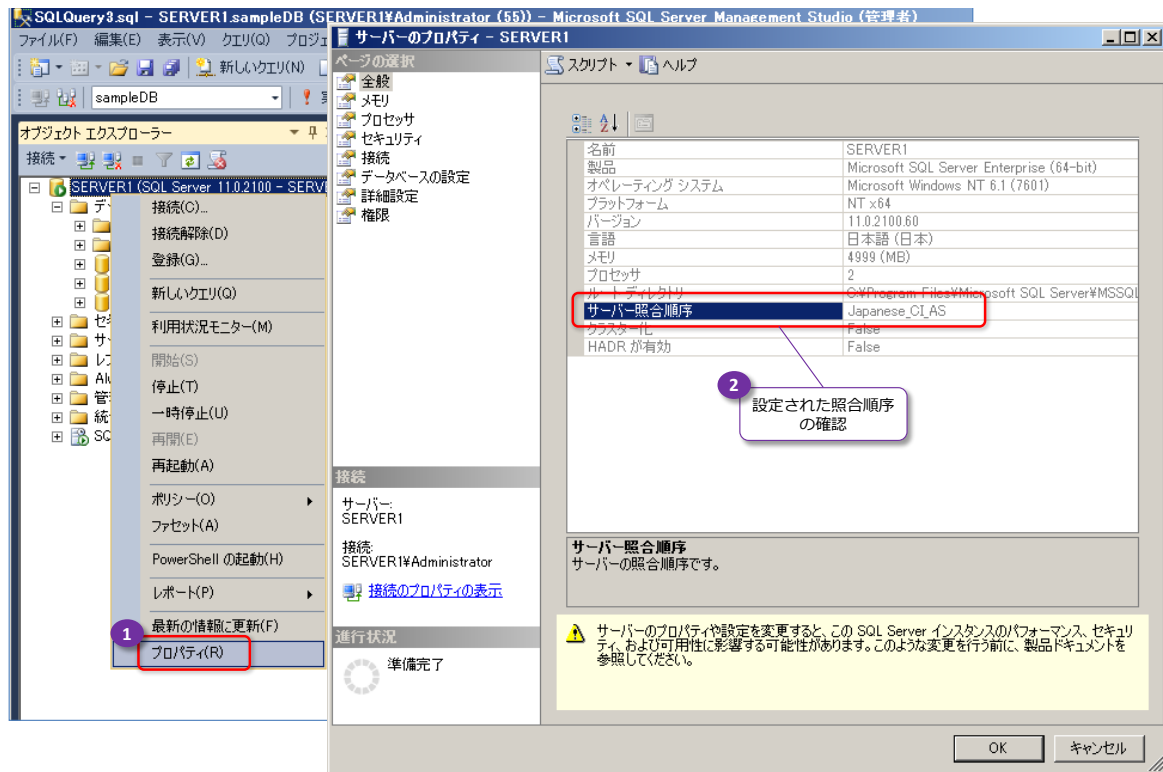
SQL Server レベルでの照合順序の設定は、SQL Server のインストール時に、次の画面で設定しています。



SQL Server 2012 の既定値は、SQL Server 2000、2005、2008、2008 R2 のデフォルトの照合順序と同じ「**Japanese_CI_AS**」に設定されています。照合順序を変更したい場合には、[**カスタマイズ**] ボタンをクリックします。

◆ 照合順序の確認

インストール時に設定した照合順序を確認するには、次のように Management Studio で SQL Server の名前を右クリックして、[**プロパティ**] をクリックします。



これにより、[サーバーのプロパティ] ダイアログの[全般] ページが表示されるので、[サーバー照合順序] 欄で確認することができます。

ただし、このダイアログでは、照合順序を変更することはできません。前述したように、サーバーの照合順序を変更したい場合は、SQL Server を再セットアップしなければなりません。

4.2 Japanese_CI_AS の動作 (SQL Server 2012 の既定値)

➡ Japanese_CI_AS の動作

SQL Server 2000/2005/2008/2008 R2/2012 のデフォルトの照合順序「Japanese_CI_AS」では、次のように動作します。

- ひらがなとカタカナを区別しない
- 半角と全角を区別しない
- 大文字と小文字を区別しない
- アクセントを区別する

➡ Let's Try

それでは、これを試してみましょう。ここでは、「sampleDB」データベース内の「emp」テーブル（以下のデータ）を利用して、照合順序を試してみましょう。

empno	empname	sal	hiredate	deptno
1	鈴木 一郎	500000	1998-04-01 00:00:00.000	10
2	小田 良夫	300000	1999-04-01 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	1996-04-01 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

➡ ひらがなとカタカナの区別は？

1. まずは、ひらがなとカタカナが区別されるかどうかを確認してみましょう。"ひらがな" のデータは、次のように「浅田 ゆかり」さんで試すことができます。

```
USE sampleDB

SELECT * FROM emp
WHERE empname = '浅田 ゆかり'
```

USE sampleDB					
SELECT * FROM emp					
WHERE empname = '浅田 ゆかり'					
100 %					
結果 メッセージ					
	empno	empname	sal	hiredate	deptno
1	3	浅田 ゆかり	NULL	NULL	20

2. 次に、検索条件の「ゆかり」を、"カタカナ" の「ユカリ」へ変更して実行してみます。

```
SELECT * FROM emp
WHERE empname = '浅田 ユカリ'
```

empno	empname	sal	hiredate	deptno
3	浅田 ゆかり	NULL	NULL	20

このようにカタカナで検索しても、ひらがなで検索したときと同じように結果を取得できます。つまり、Japanese_CI_AS 照合順序では、ひらがなとカタカナは区別されません。

➡ 半角と全角の区別は？

3. 次に、検索条件の「ユカリ」を "半角カタカナ" の「ㇿかり」へ変更して実行し、半角と全角が区別されるかどうかを確認してみましょう。

```
SELECT * FROM emp
WHERE empname = '浅田 ㇿかり'
```

empno	empname	sal	hiredate	deptno
3	浅田 ゆかり	NULL	NULL	20

半角カタカナでも同じように結果を取得できます。このように、Japanese_CI_AS 照合順序では、全角と半角も区別されません。

また、半角スペースと全角スペースも区別されないので、次のように「浅田」と「ゆかり」の間のスペースを半角から全角にしても、同じように結果を取得することができます。

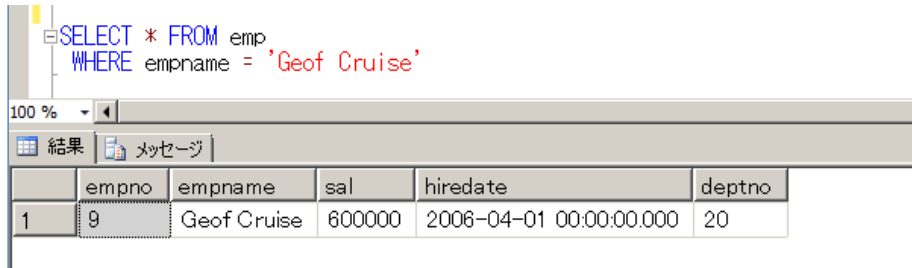
```
SELECT * FROM emp
WHERE empname = '浅田   ゆかり'
```

empno	empname	sal	hiredate	deptno
3	浅田 ゆかり	NULL	NULL	20

➡ 大文字と小文字の区別は？

4. 次に、英字の大文字と小文字を区別するかどうかを確認してみましょう。"英字" のデータは、次のように「Geof Cruise」さんで試すことができます。

```
SELECT * FROM emp
WHERE empname = 'Geof Cruise'
```



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

```
SELECT * FROM emp
WHERE empname = 'Geof Cruise'
```

Below the query window, the 'Results' tab is selected, displaying a table with the following data:

	empno	empname	sal	hiredate	deptno
1	9	Geof Cruise	600000	2006-04-01 00:00:00.000	20

5. 続いて、検索条件を「GEOF CRUISE」とすべて大文字にして検索してみます。

```
SELECT * FROM emp
WHERE empname = 'GEOF CRUISE'
```

empno	empname	sal	hiredate	deptno
9	Geof Cruise	600000	2006-04-01 00:00:00.000	20

大文字で検索しても同じように結果を取得できます。このように、Japanese_CI_AS 照合順序では、大文字と小文字も区別されません。

6. 次に、検索条件を「geOF cRUISE」と、大文字と小文字をバラバラにして検索してみましょう。

```
SELECT * FROM emp
WHERE empname = 'geOF cRUISE'
```

empno	empname	sal	hiredate	deptno
9	Geof Cruise	600000	2006-04-01 00:00:00.000	20

同じように結果を取得できることから、大文字と小文字を区別しないことを再確認できます。

7. 次に、検索条件を「Geof Cruise」とすべて全角文字にして検索してみましょう。

```
SELECT * FROM emp
WHERE empname = 'Geof Cruise'
```

empno	empname	sal	hiredate	deptno
9	Geof Cruise	600000	2006-04-01 00:00:00.000	20

こちらも同じように結果を取得できることから、半角と全角を区別しないことを再確認できます。

4.3 照合順序の種類

➡ 照合順序の種類

SQL Server 2000／2005／2008／2008 R2／2012 のデフォルトの照合順序「**Japanese_CI_AS**」の特徴を再掲すると、次のとおりです。

- ひらがなとカタカナを区別しない
- 半角と全角を区別しない
- 大文字と小文字を区別しない
- アクセントを区別する

照合順序名の **Japanese_CI_AS** の **AS** は、**Accent Sensitive** の略で、アクセントを Sensitive (区別する) という意味です。また、**CI** は、**Case Insensitive** の略で、Case は「Upper Case (大文字) と Lower Case (小文字)」、Insensitive は「区別しない」という意味です。

したがって、照合順序には、「**Japanese_CS_AS**」(Case Sensitive : 大文字と小文字を区別する) や「**Japanese_CI_AI**」(Accent Insensitive : アクセントを区別しない) などもあります。また、ひらがなとカタカナを意味する **K** (Kana) を入れた「**Japanese_CI_AS_KS**」(Kana Sensitive : ひらがなとカタカナを区別する)、半角と全角を意味する **W** (Wide) を入れた「**Japanese_CI_AS_WS**」(Wide Sensitive : 半角と全角を区別する) もあり、次のような種類があります。

大文字と小文字の区別	CI (Case Insensitive)	区別しない
	CS (Case Sensitive)	区別する
アクセントの区別	AI (Accent Insensitive)	区別しない
	AS (Accent Sensitive)	区別する
ひらがなとカタカナの区別	KI (Kana Insensitive)	区別しない
	KS (Kana Sensitive)	区別する
半角と全角の区別	WI (Wide Insensitive)	区別しない
	WS (Wide Sensitive)	区別する

➡ Japanese_BIN (バイナリ順)

照合順序には、大文字と小文字、半角と全角などを “すべて区別できる” 照合順序として、「**Japanese_BIN**」(バイナリ順 : Binary Sort) または「**Japanese_BIN2**」が用意されています。これらのバイナリの照合順序では、**漢字コード** (char や varchar データ型なら **Shift-JIS**、nchar や nvarchar データ型なら **Unicode**) での比較ができるようになります。データ型については、次の STEP で詳しく説明します。

大文字と小文字、半角と全角、ひらがなとカタカナには、それぞれ異なる漢字コードが割り当てられているので、**Japanese_BIN** または **Japanese_BIN2** を利用すれば、すべてを区別できるようになっています。

◆ Japanese_XJIS_100 : JIS2004 対応の照合順序

Windows Vista 以降の OS では、**JIS X 0213:2004 (JIS2004)** へ対応して、いくつかの漢字の“字形の変更”と“漢字の追加”が行われました。JIS2004 で追加された漢字を正しく照合（比較）するには、「**Japanese_XJIS_100**」照合順序を利用する必要があります。

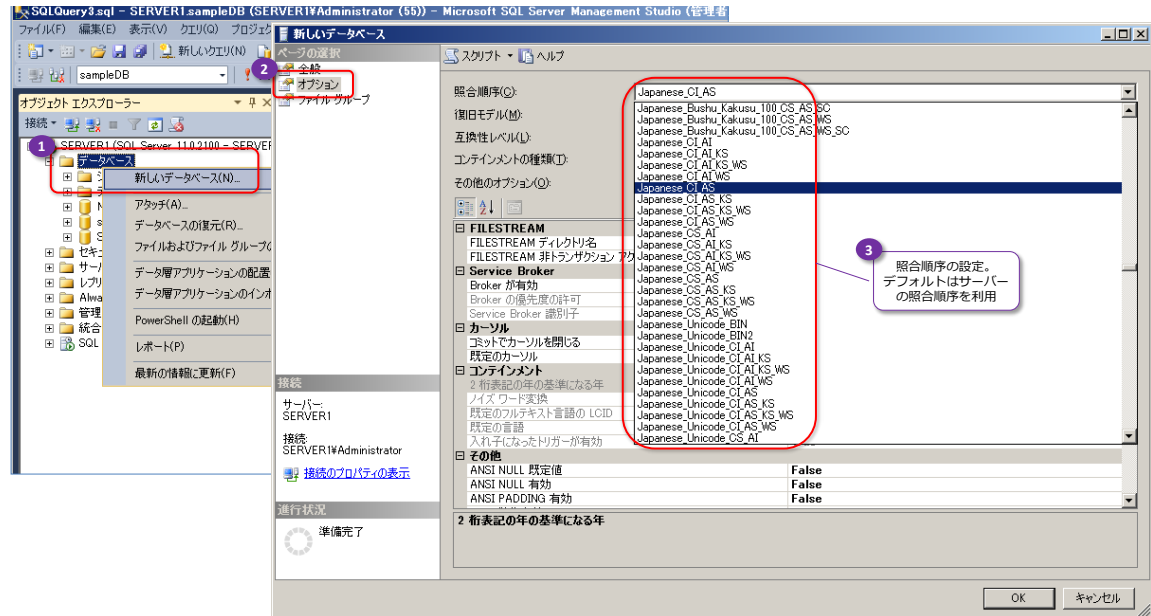
Japanese_XJIS_100 の「**100**」は、SQL Server 2008 のときの内部バージョン番号「**10.x**」という意味で、SQL Server 2005（内部バージョン番号 **9.0**）からサポートされた JIS2004 対応の照合順序「**Japanese_90**」をバージョン アップさせたものです。

JIS2004 のデータの扱いについては、STEP5 で説明しています。

4.4 データベース単位での照合順序の設定

◆ データベース単位での照合順序の設定

照合順序は、データベース単位でも設定することができます。これは、次のようにデータベース作成時の [オプション] ページで設定できます。



デフォルトは、「<既定>」に設定されるので、SQL Server のインストール時に設定したサーバーの照合順序（デフォルトは Japanese_CI_AS）が適用されます。

◆ テーブルの列単位での照合順序の設定

照合順序は、CREATE TABLE ステートメントによるテーブルの作成時に、列単位で設定することもできます。これは、次のようにデータ型の隣に、**COLLATE** 句を使って照合順序名を指定します。

```
CREATE TABLE 社員
( 社員番号 int PRIMARY KEY
, 氏名 varchar(50) COLLATE Japanese_CI_AS NOT NULL
, 給与 int NULL
)
```

◆ Let's Try

それでは、列単位での照合順序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「empTest」という名前のテーブルを作成する CREATE TABLE ステートメントを実行しましょう。**empname** 列の照合順序には「Japanese_CS_AS」と指定して、Case Sensitive（大文字と小文字を区別）するようにし

ます。

```
USE sampleDB
CREATE TABLE empTest
( empno int
, empname varchar(50) COLLATE Japanese_CS_AS )
```

2. 次に、「**emp**」テーブルの「**empno**」と「**empname**」列のデータを、「**empTest**」テーブルへ INSERT します。

```
INSERT INTO empTest
SELECT empno, empname FROM emp
```

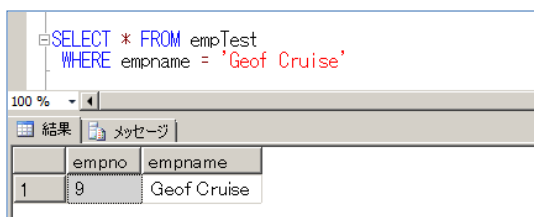
3. INSERT が完了したら、追加されたデータを確認します。

```
SELECT * FROM empTest
```

empno	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

4. 次に、大文字と小文字を区別するかどうかを確認するために、「英字」のデータを含む「**Geof Cruise**」さんで検索してみます。

```
SELECT * FROM empTest
WHERE empname = 'Geof Cruise'
```



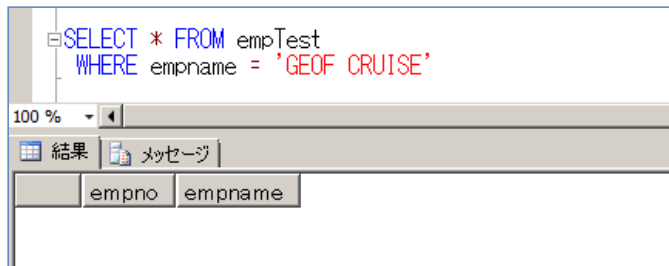
The screenshot shows the SQL Server Enterprise Manager interface. The query window displays the same SQL statement as before. Below the query window, the 'Results' pane shows a single row with empno 9 and empname Geof Cruise. The 'Messages' pane is also visible.

empno	empname
9	Geof Cruise

このように格納されているデータと大文字・小文字すべてを正しく指定した場合は、結果を取得することができます。

5. 続いて、検索条件を「**GEOF CRUISE**」とすべて大文字にして検索してみます。

```
SELECT * FROM empTest
WHERE empname = 'GEOF CRUISE'
```



結果は 1 件も表示されず、大文字と小文字を区別していることを確認できます。このように、照合順序は、列単位で変更することも可能です。

➡ 既存のテーブルの列に対して照合順序を変更する方法

上述の例は、CREATE TABLE ステートメント実行時（テーブル作成時）の照合順序の変更でしたが、既に作成済みのテーブルの列に対する照合順序の変更は、ALTER TABLE ステートメントから設定することができます。これは、次のように変更したい列のデータ型の隣に、**COLLATE** 句を使って照合順序名を指定します。

```
ALTER TABLE empTest
ALTER COLUMN
    empname varchar (50) COLLATE Japanese_CI_AS
```

Note：一時テーブルとテーブル変数の照合順序

一時テーブルとテーブル変数については、本自習書シリーズの「開発者のための Transact-SQL 応用」編で説明しますが、この 2 つの照合順序は、既定では **tempdb** システム データベースの照合順序へ設定されます。

4.5 SQL ステートメント単位の照合順序の指定

➡ SQL ステートメント単位の照合順序の指定

照合順序は、SQL ステートメント単位で指定することもできます。これは、WHERE 句の検索条件の隣へ COLLATE 句を付けて、次のように利用します。

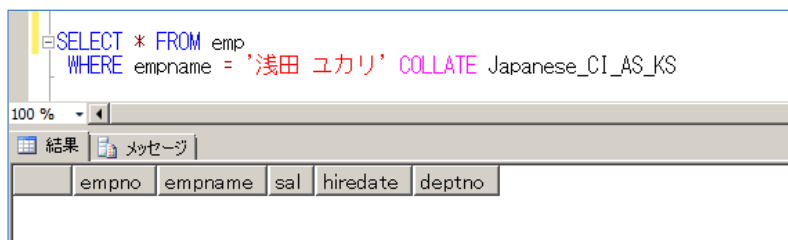
```
SELECT 選択リスト FROM テーブル名
WHERE 検索条件 COLLATE 照合順序名
```

➡ Let's Try

それでは、これを試してみましょう。

1. まずは、「emp」テーブルの「empname」列に対して、「Japanese_CI_AS_KS」照合順序を指定して、ひらがなとカタカナを区別するようにしてみましょう。

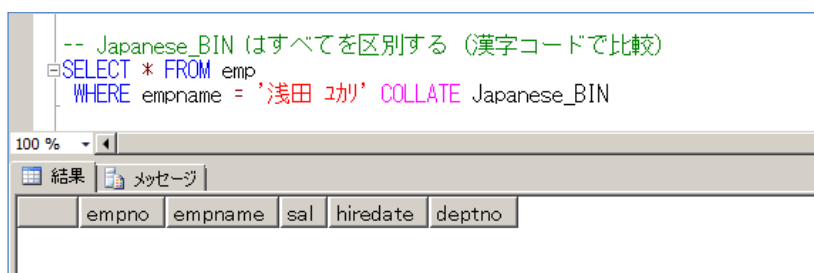
```
SELECT * FROM emp
WHERE empname = '浅田 ユカリ' COLLATE Japanese_CI_AS_KS
```



データは「浅田 ゆかり」として格納されているので、カタカナの「ユカリ」で検索した場合には、データを区別して、ヒットしなくなることを確認できます。

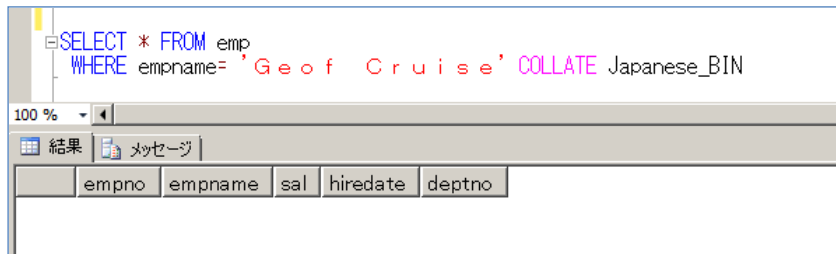
2. 次に、「Japanese_BIN」照合順序を指定して、すべてを区別することを確認してみましょう。検索条件を "半角カタカナ" の「ㇿカリ」へ変更して実行し、半角と全角が区別されることを確認します。

```
SELECT * FROM emp
WHERE empname = '浅田 ㇿカリ' COLLATE Japanese_BIN
```



3. 次に、同じく「**Japanese_BIN**」照合順序を指定して、「**Geof Cruise**」さんに対して、すべて全角で名前を入力して、結果を確認してみましょう。

```
SELECT * FROM emp
WHERE empname= 'Geof Cruise' COLLATE Japanese_BIN
```



Japanese_BIN によって半角と全角が区別されていることを確認できます。このように、照合順序は、SQL ステートメントの実行時に指定することもできます。

Note : SQL ステートメント単位の照合順序はできる限り利用しない

SQL ステートメント単位の照合順序は、内部的にはインデックスを利用しないテーブル スキャンまたはインデックス スキャン（インデックスの全スキャン）が発生するため、パフォーマンスが悪くなります。したがって、COLLATE 句を利用した SQL ステートメント単位の照合順序はなるべく使わなくて済むよう、事前に（SQL Server をインストールする前に）照合順序の設計をきちんと考慮しておく必要があります。インデックスについては、本自習書シリーズの「**インデックスの基礎とメンテナンス**」で詳しく説明しています。

Note : 照合順序の一覧を表示するには？

SQL Server で利用できる照合順序の一覧は、次の SQL を実行して確認することができます。

```
SELECT * FROM fn_helpcollations()
WHERE name LIKE '%japan%'
```

	name	description
1	Japanese_BIN	Japanese, binary sort
2	Japanese_BIN2	Japanese, binary code point comparison sort
3	Japanese_CI_AI	Japanese, case-insensitive, accent-insensitive, kanatype-insensit..
4	Japanese_CI_AI_WS	Japanese, case-insensitive, accent-insensitive, kanatype-insensit..
5	Japanese_CI_AI_KS	Japanese, case-insensitive, accent-insensitive, kanatype-sensitiv..
6	Japanese_CI_AI_KS_WS	Japanese, case-insensitive, accent-insensitive, kanatype-sensitiv..
7	Japanese_CI_AS	Japanese, case-insensitive, accent-sensitive, kanatype-insensitiv..
8	Japanese_CI_AS_WS	Japanese, case-insensitive, accent-sensitive, kanatype-insensitiv..
9	Japanese_CI_AS_KS	Japanese, case-insensitive, accent-sensitive, kanatype-sensitive,..
10	Japanese_CI_AS_KS_WS	Japanese, case-insensitive, accent-sensitive, kanatype-sensitive,..
11	Japanese_CS_AI	Japanese, case-sensitive, accent-insensitive, kanatype-insensitiv..

STEP 5. データ型

SQL Server には、色々な種類のデータ型が用意されています。この STEP では、データ型のそれぞれの特徴や利用方法について説明します。

この STEP では、次のことを学習します。

- ✓ データ型の種類
- ✓ 文字データ型 (char、varchar)
- ✓ 8,000 バイト超えの文字データ (varchar(max))
- ✓ Unicode データ型 (nchar、nvarchar)
- ✓ 整数型 (bigint、int、smallint、tinyint)
- ✓ 真数データ型 (decimal、numeric)
- ✓ 概数データ型 (real、float)
- ✓ 金額 (money、smallmoney)
- ✓ 日付データ型 (datetime、date、time)

5.1 データ型の種類

➡ データ型の種類

SQL Server 2012 がサポートしているデータ型には、次の表に挙げたとおり多くの種類がありますが、多くの方がメインで利用していくのは、文字データ格納用の「**char**」と「**varchar**」、整数データ格納用の「**int**」、小数点以下のデータを格納するための「**decimal**」または「**numeric**」、日付データを格納するための「**datetime**」や「**date**」です。

分類		データ型	使用する バイト数	説明	対応する Oracle のデータ型	対応する Access のデータ型
文字		char (n)	n	固定長文字列 (8000バイトまで)	char(n)	テキスト型
		varchar (n)	n	可変長文字列 (8000バイトまで)	varchar2(n)	(255 文字まで)
		varchar (max) text	16+a	可変長文字列 (2G/バイトまで) a はデータサイズ	long CLOB	メモ型 (65535 文字まで)
Unicode 文字		nchar (n)	n x 2~4	Unicode対応 固定長文字列 (4000文字まで)	nchar(n)	
		nvarchar (n)	n x 2~4	Unicode対応 可変長文字列 (4000文字まで)	nvarchar2(n)	
		nvarchar (max) ntext	16+a	Unicode対応 可変長文字列 (2G/バイトまで)	NCLOB	
整数型		tinyint	1	0 から 255	number(p)	バイト型
		smallint	2	-32768 から 32767	int は number(10)	整数型
		int	4	-2 ³¹ から 2 ³¹ -1	に相当	長整数型
		bigint	8	-2 ⁶³ から 2 ⁶³ -1		
小数	真数型	decimal (p, s)	5~17	p:全体桁 s:少数点以下桁	number (p, s)	十進型
		numeric (p, s)	5~17	最下位桁まで精度を保つ (2-17 バイト)		
	概数型	float (n)	4~8	精度 8-15桁	BINARY_FLOAT	単精度浮動小数点型
		real	4	精度 1-7桁	BINARY_DOUBLE	倍精度浮動小数点型
日付時刻型		smalldatetime	4	分単位の精度	date (秒単位)	日付/時刻型
		datetime	8	3.33ミリ秒単位の精度		
		date	3	日単位の精度		
		time	3~5	時間のみを格納し、100ナノ秒単位の精度	TIMESTAMP (ナノ秒単位)	
		datetime2	6~8	100ナノ秒単位の精度		
		datetimeoffset	8~10	100ナノ秒単位の精度でタイムゾーンも格納		
金額		smallmoney	4	範囲小	number (p, s)	通貨型
		money	8	範囲大		
バイナリ		binary (n)	n	固定長バイナリ データ (8000バイトまで)	raw(n)	OLE オブジェクト型
		varbinary (n)	n	可変長バイナリ データ (8000バイトまで)	long raw	
		varbinary (max) image	16+a	可変長バイナリ データ (2G/バイトまで) a はデータサイズ	BLOB	
特殊		bit	1	1 または 0		Yes/No 型
		table	a	テーブル形式のデータを格納		
		XML	a	XML データを格納可能なデータ型	Oracle XML DB	
		uniqueidentifier	16	GUID (グローバル ユニークID) を格納		
		timestamp	8	同時更新を識別するための行バージョンを格納		
		sql_variant	a	複数のデータ型を混在可能なデータ型		
		geometry geography	a	GIS (地理情報システム) における経度や緯度などの空間データを格納できるデータ型	Oracle Spatial オプション	
		FileStream	a	OS のファイル データを格納可能なデータ型	Oracle SecureFiles	

以降では、これらのデータ型の利用方法を説明します。

5.2 文字データ型：char、varchar

➡ char と varchar

char と varchar は、文字データ（Character）を格納するためのデータ型です。両者の違いは、char が“**固定長**”、varchar が“**可変長（Variable）**”のデータ型であるという点です。固定長では、指定したバイト数分の領域が使用されるのに対し、可変長では、実際のデータ分のみの領域が使用されます。

char と varchar データ型は、次のように利用します。

```
CREATE TABLE テーブル名
( 列名1 char (n)
, 列名2 varchar (n)
, ... )
```

n には、格納したいデータのサイズ（最大値）をバイト単位で指定し、**1～8,000** までの値を指定できます。

➡ Let's Try

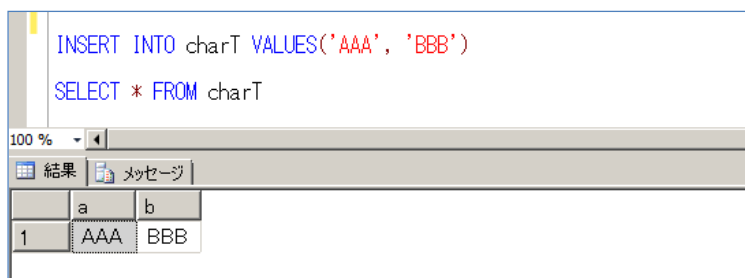
それでは、char と varchar データ型を試してみましょう。

1. まずは、次のように「sampleDB」データベース内へ「charT」という名前のテーブルを作成し、「a」列を「char(5)」、「b」列を「varchar(5)」として、5 バイトの文字データが格納できるようにします。

```
USE sampleDB
CREATE TABLE charT
( a char (5)
, b varchar (5) )
```

2. 次に、char と varchar データ型へ文字データを INSERT してみましょう。

```
INSERT INTO charT VALUES('AAA', 'BBB')
SELECT * FROM charT
```

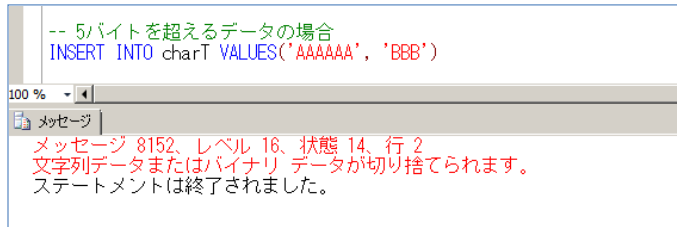


```
INSERT INTO charT VALUES('AAA', 'BBB')
SELECT * FROM charT
```

	a	b
1	AAA	BBB

3. 続いて、次のように 5 バイトを超えるデータを「a」列へ格納してみましょう。

```
INSERT INTO charT VALUES('AAAAAA', 'BBB')
```



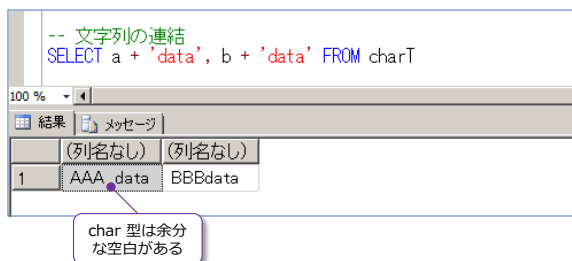
結果は「文字列データまたはバイナリ データが切り捨てられます」エラーが表示されて、データの格納が失敗します。char と varchar データ型では、指定サイズを超えた場合には、このエラーが発生します。

➡ char と varchar の違い

次に、char と varchar データ型の違いを確認してみましょう。

1. char 型の「a」列と varchar 型の「b」列に対して、文字列を連結できる演算子「+」を利用して、「data」という文字列を連結してみましょう。

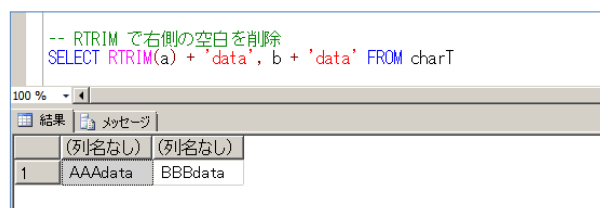
```
SELECT a + 'data', b + 'data' FROM charT
```



結果は、char データ型の場合は、データ「AAA」と文字列「data」との間に余分な空白が入っていることを確認できます。char データ型の「a」列は、**char(5)** と定義しているので、5 バイトに満たない分の空白（AAA は 3 バイトなので 2 バイト分の空白）があるため、このような結果になっています。

Note : 空白を取り除く RTRIM 関数

詳しくは STEP6 で説明しますが、「RTRIM」という関数を利用すると、余分な空白を削除して結果を取得することができます。たとえば、次のように RTRIM 関数を利用すれば、a 列と文字列「data」との間の空白を取り除くことができます。



➡ WHERE 句の条件式での char 型の空白の扱い

WHERE 句の条件式で char データ型の列が指定された場合は、右側に追加された余分な空白は無視されます。

1. これは、次のように試すことができます。

```
SELECT * FROM charT
WHERE a = 'AAA'
```

-- WHERE 句では、右側の空白が無視される

```
SELECT * FROM charT
WHERE a = 'AAA'
```

100 %

結果 メッセージ

	a	b
1	AAA	BBB

「AAA」データには、右側に余分な空白がありますが、「AAA」のように空白を付けて条件式を記述しなくても、結果を取得することができます。

Note：空文字と NULL の違い (Oracle との相違点)

Oracle では、" (空の文字列) は NULL 値として扱われますが、SQL Server では、空文字と NULL 値は区別して扱われます。

```
CREATE TABLE nullIT
( a int NOT NULL
, b varchar(50) NULL )
INSERT INTO nullIT VALUES ( 1, '' )
INSERT INTO nullIT VALUES ( 2, NULL )
```

'' (空の文字列)

NULL 値

= '' で検索すると空文字のみヒット

```
SELECT * FROM nullIT WHERE b = ''
```

	a	b
1	1	

IS NULL で検索すると NULL 値のみヒット

```
SELECT * FROM nullIT WHERE b IS NULL
```

	a	b
1	2	NULL

SQL Server では、空文字と NULL 値を区別する

Note：char と varchar の使い分け

char と varchar は、パフォーマンスが良いのは char、ディスクの使用効率が良いのは varchar です。したがって、ほぼ固定の長さの文字データ（都道府県名や性別、文字を含む商品コードなど）を格納する場合には char 型を選択し、データによって長さが可変で、バラツキが大きいもの（商品の説明や、住所、プロフィールなど）を格納する場合には varchar 型を選択すると良いでしょう。

5.3 8,000 バイト超えの文字データ： varchar(max)

➡ varchar(max) と text データ型

char と varchar データ型の最大サイズは、**8,000 バイト**までなので、8,000 バイトを超えるデータを格納したい場合には、**varchar(max)** データ型を利用します。このデータ型は、SQL Server 2000 以前のバージョンの **text** データ型を機能向上させたものです。SQL Server 2012 でも text データ型は残っていますが、あくまでも下位互換のために残っているだけで、varchar(max) データ型よりも多くの制限を受けます (text データ型では、STEP6 で紹介する文字列操作の関数で多くの制約を受けます)。したがって、8,000 バイト超えのデータを扱う場合には、varchar(max) データ型を利用することをお勧めします。

➡ Let's Try

それでは、これを試してみましょう。

1. varchar(max) データ型は、char や varchar データ型と同じように試すことができます。次のように「**sampleDB**」データベース内へ「**charT2**」という名前のテーブルを作成し、「**a**」列を「**varchar(max)**」として文字データを格納できるようにします。

```
USE sampleDB
CREATE TABLE charT2
( a varchar(max) )

INSERT INTO charT2 VALUES('AAA')

SELECT * FROM charT2
```

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

```
USE sampleDB
CREATE TABLE charT2
( a varchar(max) )

INSERT INTO charT2 VALUES('AAA')

SELECT * FROM charT2
```

Below the script, the 'Results' tab is selected, displaying a table with one row and one column:

	a
1	AAA

このように varchar(max) データ型は、char や varchar データ型と同じように利用することができ、かつ 8,000 バイトを超えるデータを格納することができます。

5.4 Unicode データ型： nchar、nvarchar

➡ nchar、nvarchar、nvarchar(max)

SQL Server には、Unicode 文字を格納するためのデータ型として、**nchar** と **nvarchar**、**nvarchar(max)**、**ntext** が用意されています。これらは、char や varchar、text データ型の先頭に「n」（National の略）を付けただけで、違いは Unicode データを格納できるかどうかだけです。

Unicode データ型は、次のように利用します。

```
CREATE TABLE テーブル名
( 列名1 nchar (n)
, 列名2 nvarchar (n)
, 列名3 nvarchar (max)
, ... )
```

n には、格納したいデータの文字数（最大値）を指定し、**1～4,000** までの値を指定できます。char と varchar データ型では、n をバイト数で指定したのに対して、Unicode データ型では、文字数で指定することに注意してください。n へ指定できる最大値は 4,000 文字で、char や varchar 型での最大値 8,000 バイトとは異なりますが、これは、Unicode データが 1 つの文字で 2～4 バイトを消費するためです。

4,000 文字を超える Unicode データを格納したい場合には、**nvarchar(max)** と **ntext** データ型が用意されています。ntext は下位互換用のデータ型なので、nvarchar(max) を利用することをお勧めします。

➡ N プレフィックス

Unicode データ型に対して、Unicode データを格納する場合には、次のようにプレフィックス（接頭辞）として「**N**」を付ける必要があります。

```
INSERT INTO テーブル名 VALUES ( N'Unicodeデータ', ... )
```

➡ Let's Try

それでは、Unicode データ型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「**uniT**」という名前のテーブルを作成し、「**b**」列のデータ型には **varchar(100)** を、「**c**」列のデータ型には **nvarchar(50)** を指定します。

```
USE sampleDB
CREATE TABLE unit
(
  a int
, b varchar(100)
, c nvarchar(50) )
```

2. 次に、作成した「**uniT**」テーブルへ、Unicode にしかない文字として、風月堂の「**風**」（ふう）を INSERT してみましょう。「風」は、通常の IME 変換では一覧へ出てこない文字なので、サンプル スクリプト内の「**Step5_Query.sql**」ファイルからスクリプトをコピーするか、後述の Note で紹介する「IME パッドの手書きツール」を利用して、入力してください。

```
-- Unicode データ「風」を追加
INSERT INTO uniT VALUES ( 1, '風月堂', '風月堂' )
INSERT INTO uniT VALUES ( 2, '風月堂', N'風月堂' )

SELECT * FROM uniT
```

```

USE sampleDB
CREATE TABLE uniT
(
  a int
 ,b varchar(100)
 ,c nvarchar(50) )

INSERT INTO uniT VALUES ( 1, '風月堂', '風月堂' )
INSERT INTO uniT VALUES ( 2, '風月堂', N'風月堂' )

SELECT * FROM uniT

```

00 %

結果 メッセージ

	a	b	c
1	1	?月堂	?月堂
2	2	?月堂	風月堂

varchar 列

N を付けない

N を付ける

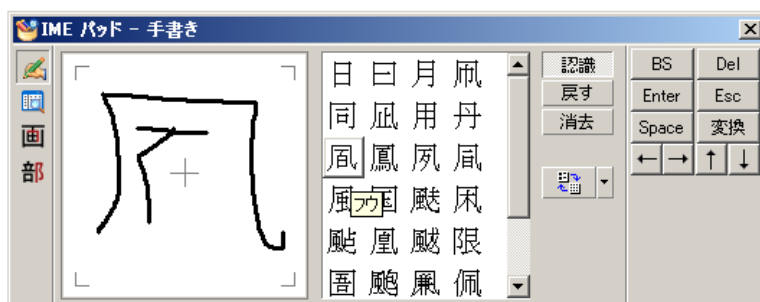
N を付けない場合と varchar データ型の場合は Unicode 文字が「?」（不明な文字）として登録される

N を付けて、nvarchar データ型の場合は Unicode 文字を正しく格納できる

N プレフィックスを付けない場合と、varchar データ型の場合は、Unicode 文字が「?」（不明な文字）として登録されてしまうことを確認できます。Unicode 文字を扱う場合は、N プレフィックスを付け忘れないように注意しましょう。

Note : IME パッドの手書きツールから「風」を入力

「風」を入力するときは、次のように IME パッドの手書きツールを利用することもできます。



◆ その他の Unicode 文字

3. 次に、そのほかの Unicode 文字として、登録商標の「®」やトレード マーク「™」、著作権 (コピー ライト) の「©」、中国語の「你好」(ニイハオ)、ハングルの「안녕하세요」(アンニョンハセヨ)などを「uniT」テーブルへ追加してみましょう。これらの文字も通常の IME 変換では出てこない文字なので、サンプル スクリプト内の「Step5_Query.sql」ファイルからスクリプトをコピーして実行してみてください。

```
INSERT INTO uniT VALUES ( 4, 'SQL Server®', N'SQL Server®' )
INSERT INTO uniT VALUES ( 3, 'Windows Azure™', N'Windows Azure™' )
INSERT INTO uniT VALUES ( 5, '©Microsoft', N'©Microsoft' )
INSERT INTO uniT VALUES ( 6, '你好', N'你好' )
INSERT INTO uniT VALUES ( 7, '안녕하세요', N'안녕하세요' )

SELECT * FROM uniTest
```

The screenshot shows a SQL script being executed in SQL Server Enterprise Manager. The script inserts five rows into the 'uniT' table, each with a unique identifier (a), a string (b), and a Unicode string (c). The results are displayed in a table below the script.

Callouts from the script and results table:

- 登録商標「®」 (Registered Trademark symbol)
- トレードマーク「™」 (Trademark symbol)
- 著作権の「©」 (Copyright symbol)
- 中国語の「你好」(ニイハオ) (Chinese 'Hello')
- ハングルの「안녕하세요」(アンニョンハセヨ) (Korean 'Hello')
- nvarchar なら Unicode 文字を格納できる (With nvarchar, Unicode characters can be stored)
- varchar では、Unicode 文字が「?」(不明な文字)として登録される (With varchar, Unicode characters are registered as '?') (Unknown character)

	a	b	c
1	1	?月堂	?月堂
2	2	?月堂	風月堂
3	4	SQL ServerR	SQL Server®
4	3	Windows Azure?	Windows Azure™
5	5	cMicrosoft	©Microsoft
6	6	?好	你好
7	7	?????	안녕하세요

Note : トレード マークや商標登録、著作権マークの入力

トレード マークや商標登録、著作権マークは、Microsoft Word を利用すると、簡単に入力できます。Microsoft Word では、「Ctrl+Alt+R」キーで「®」、「Ctrl+Alt+T」キーで「™」、「Ctrl+Alt+C」キーで「©」を入力することができます。

◆ Unicode データ型の使いどころ

これまで試してきたように、Unicode 文字をデータベース内へ格納したい場合には、Unicode データ型 (nchar や nvarchar、nvarchar(max)) を利用する必要があります。逆に言うと、Unicode 文字を格納しなくても良い場合は (Shift-JIS コード内の文字のみを格納する場合は)、char や varchar データ型を利用します。

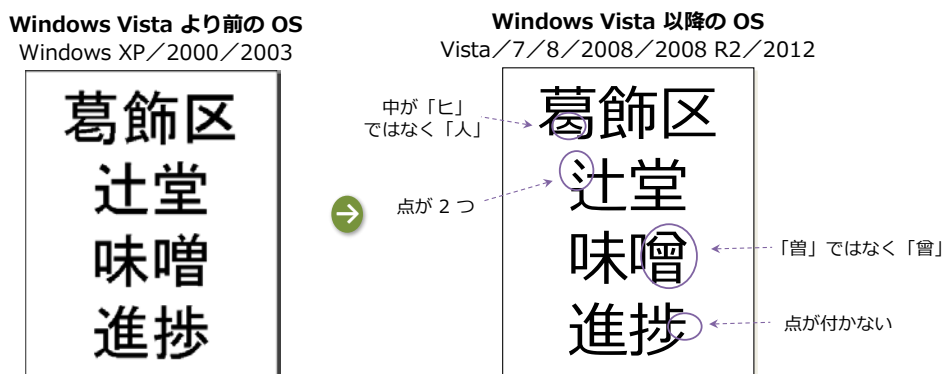
Note : Unicode データ型の注意点

Unicode データ型は、1 文字あたり 2~4 バイトを消費し、内部的には 1 バイトで済む英数字のデータ (A、B、C、1、2 など) を格納する場合でさえも 2 バイトを消費します。したがって、Unicode データ型を多用すると、データ サイズ (データベース サイズ) が大きくなってしまい、ディスク I/O (ディスクからの読み取りと書き込み) が増えることになるので、パフォーマンス的には余分なオーバーヘッドが発生します。したがって、何でもかんでも Unicode データ型を利用するというのは避け、必要な部分にのみ利用するようにしましょう。

なお、SQL Server 2008 R2 からは、「Unicode 圧縮」という機能が提供されて、英数字データを圧縮 (2 バイト利用するのではなく、1 バイト使用) することもできるようになっています。

➡ JIS2004 対応のデータ型は nchar と nvarchar、nvarchar(max)

Windows Vista 以降の OS では、**JIS X 0213:2004 (JIS2004)** へ対応して、いくつかの漢字の“**字形の変更**”と“**漢字の追加**”が行われました。字形が変わった漢字には、葛飾区の「葛」や味噌の「噌」などがあります。



これらの字形が変わった漢字は、内部的な文字コードは同じなので、SQL Server へ格納したとしても、Windows Vista 以降のマシン (Windows Vista や Windows 7/8、Windows Server 2008/2008 R2/2012) からは右のように見え、それより前の OS (Windows XP や Windows 2000、Windows Server 2003 など) からは左のように見えます。文字化けが発生するわけではなく、古い字形で見えます。

新しく追加された漢字には、第3水準/第4水準漢字など 3,000 字近くがあります。これらの漢字の多くは、Unicode の“**補助文字**”を使って実現しています。補助文字は、サロゲート ペアと呼ばれる方法で、Unicode 文字の未定義領域を 2 文字分 (ペアを) 使って 1 つの文字を表現したものです。したがって、JIS2004 の新漢字の多く (補助文字) を SQL Server で扱うには、Unicode 文字を格納できるデータ型 (nchar や nvarchar) を利用する必要があります。

➡ JIS2004 対応の照合順序は Japanese_XJIS_100

STEP4 で説明したように、JIS2004 に対応した照合順序は **Japanese_XJIS_100** (SQL Server 2005 での Japanese_90 をバージョン アップさせたもの) です。この照合順序を利用しない場合は、補助文字 (サロゲート ペア) に対する文字列比較や LIKE 演算が正しく動作しません。

Note : JIS2004 の補助文字に対する文字列関数の注意点

JIS2004 の漢字のうち、補助文字を利用した漢字に対しては、文字列操作のための関数 (RIGHT や SUBSTRING など) の動作に注意する必要があります (文字列操作の関数については、STEP6 で説明します)。補助文字 (サロゲート ペア) は、前述したように内部的には 2 文字分 (4 バイト) を利用しているので、たとえば、文字列の長さを取得できる「LEN」という関数を補助文字に対して利用すると、結果は 2 倍 (2 文字分の 4 バイト) のサイズで返ってきます。これらについての詳細は、SQL Server 2005 での情報になりますが、以下の資料に記載されていますので、一読しておくことをお勧めします (資料内の照合順序 Japanese_90 を Japanese_XJIS_100 へ置き換えて読むと SQL Server 2012 対応になります)。

SQL Server における JIS2004 対応について寄せられる質問と回答

<http://support.microsoft.com/kb/931785>

SQL Server の JIS2004 対応に関するガイドライン

<http://go.microsoft.com/?linkid=6302906>

5.5 整数型：tinyint、smallint、int、bigint

◆ tinyint、smallint、int、bigint

int と付くデータ型は、いずれも整数 (integer) データを格納するためのデータ型です。それぞれの違いは、内部的に確保される領域のサイズと扱えるデータの範囲です。

データ型	内部的な 使用バイト数	扱えるデータの範囲
tinyint	1	2^8 通り = 256 通り。0 ~ 255
smallint	2	2^{16} 通り = 65536 通り。-32,768 から 32,767
int	4	2^{32} 通り = -2,147,473,648 ~ 2,147,473,647
bigint	8	2^{64} 通り = -9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807

◆ Let's Try

それでは、これを試してみましょう。

- まずは、次のように「sampleDB」データベース内へ「intT」という名前のテーブルを作成し、「a」列のデータ型を **tinyint**、「b」列のデータ型を **smallint**、「c」列のデータ型を **int**、「d」列のデータ型を **bigint** へ指定するようにします。

```
USE sampleDB
CREATE TABLE intT
( a tinyint, b smallint, c int, d bigint )

INSERT INTO intT VALUES ( 1, 1, 1, 1 )
SELECT * FROM intT
```

a	b	c	d
1	1	1	1

- 次に、tinyint データ型の「a」列へ、格納できるデータの範囲外となる“-1”を INSERT し、エラーが発生することを確認してみます。次のように入力して、実行します。

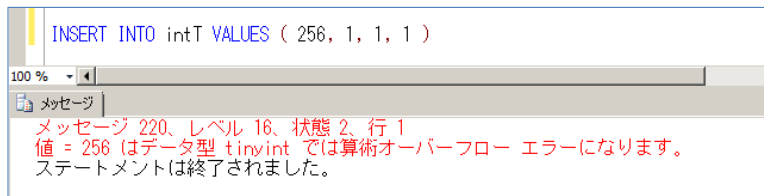
```
INSERT INTO intT VALUES ( -1, 1, 1, 1 )
```

メッセージ 220、レベル 16、状態 2、行 1
値 -1 はデータ型 tinyint では算術オーバーフロー エラーになります。
ステートメントは終了されました。

結果は、「**算術オーバーフローエラー**」が発生します。int 系のデータ型では、範囲外のデータが入力された場合には、このエラーが発生します。

3. 次に、tinyint データ型の「a」列へ、データの範囲外となる値として、今度は“**256**”を入力してみます。

```
INSERT INTO intT VALUES ( 256, 1, 1, 1 )
```



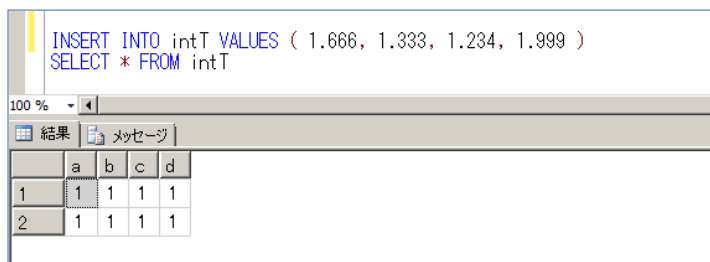
結果は、同じエラー（算術オーバーフロー）が発生したことを確認できます。このように tinyint データ型を利用する場合は、0～255 までの範囲の整数のみしか格納することができません。

➡ 小数点以下の数値は切り捨てられる

int 系のデータ型は、整数（integer）データの格納用なので、小数点付きの数が入力された場合には、小数点以下が切り捨てられて格納されます。それでは、これを試してみましょう。

4. 次のように、各列のデータへ小数点以下の数値を指定して INSERT してみます。

```
INSERT INTO intT VALUES ( 1.666, 1.333, 1.234, 1.999 )
SELECT * FROM intT
```



結果は、小数点以下の値がすべて切り捨てられて、1 として格納されていることを確認できます。

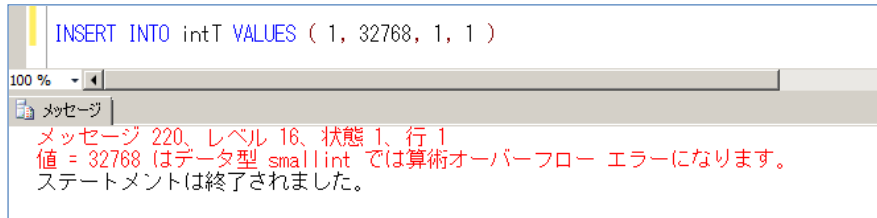
➡ smallint、int、bigint

次に、smallint と int、bigint の違いを試してみましょう。この 3 つは、マイナスの値も扱えます。前出の表のように、smallint は -32,768 から 32,767、int は -2,147,483,648 から 2,147,483,647 までのデータの範囲を扱えます。

それでは、これを試してみましょう。

5. 次のように smallint データ型の「b」列へ、データの範囲を超えた「32768」を入力して、エラーが発生することを確認してみましょう。

```
INSERT INTO intT VALUES ( 1, 32768, 1, 1 )
```



6. 続いて、smallint データ型の「b」列へ、データの範囲内の「-32768」を入力してみます。

```
INSERT INTO intT VALUES ( 1, -32768, 1, 1 )
SELECT * FROM intT
```

	a	b	c	d
1	1	1	1	1
2	1	1	1	1
3	1	-32768	1	1

これは、エラーが発生せずに正しく格納できます。

7. 次に、int データ型の「c」列へ、「32768」を入力してみます。

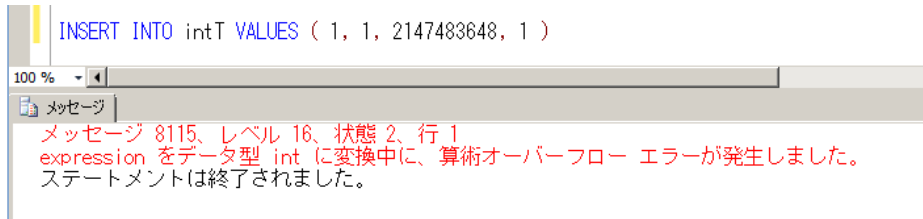
```
INSERT INTO intT VALUES ( 1, 1, 32768, 1 )
SELECT * FROM intT
```

	a	b	c	d
1	1	1	1	1
2	1	1	1	1
3	1	-32768	1	1
4	1	1	32768	1

int データ型では、「32768」はデータの範囲内なので、エラーにならないことを確認できます。

8. 続いて、int データ型の「c」列へ、データの範囲外の「2,147,483,648」を入力して、エラーが発生することを確認します。

```
INSERT INTO intT VALUES ( 1, 1, 2147483648, 1 )
```



9. 次に、bigint データ型の「d」列へ「2,147,483,648」を入力してみます。

```
INSERT INTO intT VALUES ( 1, 1, 1, 2147483648 )
SELECT * FROM intT
```

INSERT INTO intT VALUES (1, 1, 1, 2147483648)
SELECT * FROM intT

	a	b	c	d
1	1	1	1	1
2	1	1	1	1
3	1	-32768	1	1
4	1	1	32768	1
5	1	1	1	2147483648

この値は、bigint データ型では、範囲内のデータなので、エラーにはなりません。

このように、int 系のデータ型の違いは、内部的に確保される領域のサイズと扱えるデータの範囲です。

5.6 真数データ型： decimal、numeric

➡ decimal、numeric

decimal と **numeric** は、小数点付きの数値データを格納できるデータ型です。内部的には、どちらも同じものなので、どちらを利用してもかまいません。 decimal は「小数の」「10 進法の」、numeric は「数値」という意味です。

decimal と numeric は、指定したデータの桁数分の精度が保証されるので、真数データ型とも呼ばれています。桁数は、次のように指定します。

```
decimal (p, s)
または
numeric (p, s)
```

p には全体の桁数を、**s** には小数点以下の桁数を指定します (p の最大桁数は 38、s は p 以下の桁数を指定)。

➡ Let's Try

それでは、これを試してみましょう。

1. まずは、「**sampleDB**」データベース内へ「**deciT**」という名前のテーブルを作成し、「**a**」列のデータ型を **decimal(10,3)** へ指定して、「**1.66666**」というデータを INSERT します。

```
USE sampleDB
CREATE TABLE deciT ( a decimal(10,3) )
INSERT INTO deciT VALUES ( 1.66666 )
SELECT * FROM deciT
```

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

```
USE sampleDB
CREATE TABLE deciT ( a decimal(10,3) )
INSERT INTO deciT VALUES ( 1.66666 )
SELECT * FROM deciT
```

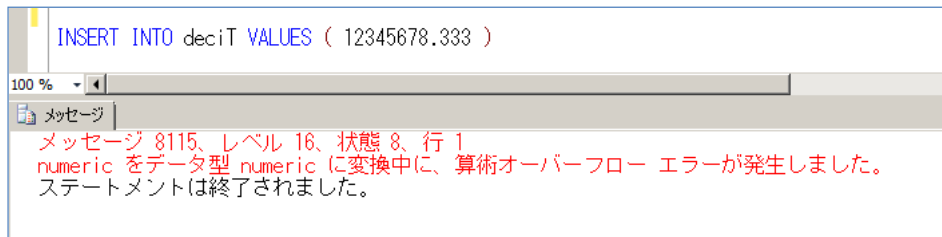
Below the query window, the 'Results' pane shows the output of the SELECT statement:

	a
1	1.667

このように、“**decimal(10,3)**”と定義した場合は、全体の桁数が **10**、小数点以下の桁数が **3** までの精度が保証されます。データが「**1.66666**」の場合は、小数点以下 3 桁で丸められて「**1.667**」として格納されます。

2. 次に、整数部の精度「**10-3=7 桁**」を超えた大きさのデータ「**12345678.333**」を INSERT してみましょう。

```
INSERT INTO deciT VALUES ( 12345678.333 )
```



このように decimal または numeric データ型では、整数部の精度を超えた場合には、「**算術オーバーフロー**」エラーが発生して、データを格納することはできません。

5.7 概数データ型：real、float

➡ real、float

real と **float** は、decimal や numeric と同じように小数点付きの数値を格納できるデータ型です。decimal や numeric との違いは、小数点以下の桁数を指定することができない点です。このため、real と float は、概数データ型とも呼ばれます。real と float の違いは、内部的な使用バイト数と精度です。real は 4 バイト、float は 8 バイトを使用して、その範囲内で扱えるデータを格納できます（Access でいう**単精度浮動小数点**が real、**倍精度浮動小数点**が float 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」データベース内へ「realT」という名前のテーブルを作成し、「a」列のデータ型を **real**、「b」列のデータ型を **float** へ指定するようにして、**1.66666...**（6を15桁分）を各列へ INSERT してみます。

```
USE sampleDB
CREATE TABLE realT ( a real, b float )
INSERT INTO realT VALUES ( 1.6666666666666666, 1.6666666666666666 )
SELECT * FROM realT
```

The screenshot shows the SQL Server Enterprise Manager interface. The top pane displays the executed SQL script. The bottom pane shows the results of the query in a table format.

	a	b
1	1.666667	1.66666666666667

結果は、real は全体で **7 桁**、float は全体で **15 桁**に丸められて格納されていることを確認できます。

5.8 金額：smallmoney、money

➡ smallmoney、money

smallmoney と **money** は、通貨データを格納できるデータ型です。違いは、内部的な使用バイト数と扱えるデータの範囲で、smallmoney は 4 バイト、money は 8 バイトです。

データ型	内部的な 使用バイト数	扱えるデータの範囲
smallmoney	4	-214,747.3648 ～ 214,747.3647
money	8	-922,337,203,685,477.5808 ～ 922,337,203,685,477.5807

smallmoney では、21 万 4748 円までしか扱えないことに注意してください。整数型の int や bigint と比べて、smallmoney と money では小数点以下を 4 桁使う分、整数部の桁数が 4 桁分少なくなっているためです。米国では、「～ドル.～セント」という形で、ドル以下を小数点で表すので、このようになっています。

したがって、日本の通貨の場合は、小数点以下はないので、smallmoney と money の代わりに、int や bigint、decimal を利用してもかまいません。

➡ Let's Try

それでは、これを試してみましょう。

1. まずは、「sampleDB」データベース内へ「monT」という名前のテーブルを作成し、「a」列のデータ型を **smallmoney**、「b」列のデータ型を **money** へ指定するようにして、「9000」というデータを INSERT します。

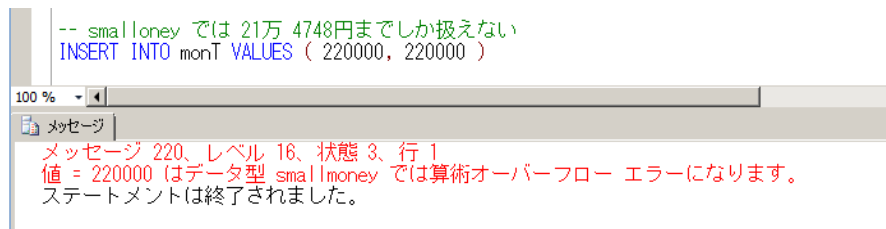
```
USE sampleDB
CREATE TABLE monT ( a smallmoney, b money )
INSERT INTO monT VALUES ( 9000, 9000 )
SELECT * FROM monT
```

	a	b
1	9000.00	9000.00

smallmoney と money では、
小数点以下 4 桁が内部使用される

2. 次に、金額を「220,000」（22 万円）にして、データを INSERT してみます。

```
INSERT INTO monT VALUES ( 220000, 220000 )
```



結果は、算術オーバーフロー エラーが発生します。前述したように smallmoney データ型では、21 万 4748 円以上のデータは扱えないからです。

5.9 日付データ型: datetime、date、time、datetime2、datetimeoffset

➡ 日付データ型

SQL Server 2012 で日付を格納できるデータ型には、次の 6 種類があり、それぞれの違いは、内部的な使用バイト数と格納できるデータの範囲（と時間の単位）です。

データ型	単位	使用バイト数	範囲
datetime	0.333 秒	8 バイト	1753-1-1 ~9999-12-31
smalldatetime	1分	4 バイト	1900-1-1 ~2079-6-6
date	日	3 バイト	1-1-1 ~9999-12-31
time	100 ナノ秒	time(7) は 5 バイト	00:00:00 ~23:59:59.9999999
datetime2	100 ナノ秒	datetime2(7) は 8 バイト	1-1-1 ~9999-12-31
datetimeoffset	100 ナノ秒	datetimeoffset(7) は 10 バイト	1-1-1 ~9999-12-31 +/-14:00

それぞれのデータ型の特徴は、次のとおりです。

datetime データ型は、1/300 秒（0.333 秒）単位の日付と時刻を扱えます。

smalldatetime データ型は、「分」単位でデータを格納できるデータ型で、使用バイト数を 4 バイトに抑えることができます。

date データ型は、「日」単位でデータを格納できるデータ型で、使用バイト数を 3 バイトに抑えることができます。時刻を格納する必要のないデータの場合には、大変便利なデータ型です。

time データ型は、「時刻」のみを格納できるデータ型で、datetime データ型よりも精度が高い時刻（100 ナノ秒 = 10 のマイナス 7 乗、小数点以下 7 桁）まで格納できるのが特徴です。

datetime2 データ型は、time データ型と同様、時刻を 100 ナノ秒まで格納でき、かつ datetime データ型のように日付も格納できるのが特徴です。

datetimeoffset データ型は、datetime2 データ型の格納範囲に加えて、タイムゾーンのオフセット（協定世界時からの時間差）を格納できるデータ型です。

SQL Server 2005 以前のバージョンまでは、日付データ型は、datetime と smalldatetime の 2 種類のみでしたが、SQL Server 2008 からは date と time、datetime2、datetimeoffset の 4 種類が追加されました。

➡ 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「dateT」という名前のテーブルを作成し、「a」列のデータ型を **datetime**、「b」列を **smalldatetime**、「c」列を **date**、「d」列を **time** と指定します。

```
USE sampleDB
CREATE TABLE dateT ( a datetime, b smalldatetime, c date, d time )
```

2. 続いて、現在時刻を取得できる **GETDATE** 関数を利用して、各列へデータを INSERT します（関数については、STEP6 で説明します）。

```
INSERT INTO dateT
VALUES ( GETDATE(), GETDATE(), GETDATE(), GETDATE() )
SELECT * FROM dateT
```

	a	b	c	d
1	2012-07-04 02:11:52.617	2012-07-04 02:12:00	2012-07-04	02:11:52.6170000

datetime は 0.333 秒単位

smalldatetime は 分単位

date は 日単位

time は 時刻のみ。100ナノ秒 (10⁻⁷、小数点以下 7桁) 単位

このように、datetime は 1/300 秒 (0.333 秒) 単位の日付と時刻を、smalldatetime は分単位、date は日単位、time は時刻のみを格納できることを確認できます。

◆ datetime2、datetimeoffset (SYSDATETIME、SYSDATETIMEOFFSET)

GETDATE 関数の精度は、0.333 秒単位で datetime データ型へ現在時刻を格納するための関数ですが、SQL Server 2008 からの新しいデータ型の time や datetime2、datetimeoffset など、100 ナノ秒 (10⁻⁷) 単位のデータ型へ現在時刻を格納するための関数として **SYSDATETIME** と **SYSDATETIMEOFFSET** (両者の違いはタイムゾーン オフセットが含まれるかどうか) があります。

それでは、これを試してみましょう。

1. 次のように、「**dateT2**」テーブルを作成し、「**a**」列のデータ型を **datetime**、「**b**」列を **time**、「**c**」列を **datetime2**、「**d**」列を **datetimeoffset** と指定し、SYSDATETIME と SYSDATETIMEOFFSET 関数を利用して現在時刻を INSERT します。

```
CREATE TABLE dateT2 ( a datetime, b time, c datetime2, d datetimeoffset )
INSERT INTO dateT2
VALUES ( SYSDATETIME(), SYSDATETIME(), SYSDATETIME(), SYSDATETIME() )
INSERT INTO dateT2
VALUES ( SYSDATETIMEOFFSET(), SYSDATETIMEOFFSET(),
        SYSDATETIMEOFFSET(), SYSDATETIMEOFFSET() )
SELECT * FROM dateT2
```

```

USE sampleDB
CREATE TABLE dateT2 ( a datetime, b time, c datetime2, d datetimeoffset )
INSERT INTO dateT2
VALUES ( SYSDATETIME(), SYSDATETIME(), SYSDATETIME(), SYSDATETIME() )
INSERT INTO dateT2
VALUES ( SYSDATETIMEOFFSET(), SYSDATETIMEOFFSET(),
        SYSDATETIMEOFFSET(), SYSDATETIMEOFFSET() )
SELECT * FROM dateT2

```

	a	b	c	d
1	2012-07-04 02:13:42.697	02:13:42.6953125	2012-07-04 02:13:42.6953125	2012-07-04 02:13:42.6953125 +00:00
2	2012-07-04 02:13:42.697	02:13:42.6972657	2012-07-04 02:13:42.6972657	2012-07-04 02:13:42.6972657 +09:00

datetime は 0.333 秒単位
time は 100ナノ秒 (10^{-7}) 単位
datetime2 は 100ナノ秒単位で格納できる。
datetimeoffset はタイムゾーンオフセットを含み、100ナノ秒単位で格納できる

time データ型では、100 ナノ秒 (10^{-7} 、小数点以下 7 桁) 単位でデータが格納されていることを確認できます。また、datetime2 と datetimeoffset データ型についても、100 ナノ秒 (10^{-7}) 単位でデータを格納できます。

datetime2 と datetimeoffset データ型の違いは、タイムゾーン オフセット (グリニッジ標準時から時間差：東京は +9 時間) を含むかどうかです。SYSDATETIME と SYSDATETIMEOFFSET 関数の違いも同様で、タイムゾーン オフセットを含むかどうかです。

Note : SYSUTCDATETIME で UTC 時刻の取得

SQL Server 2012 には、UTC 時刻 (協定世界時：Universal Time, Coordinated) を取得できる **SYSUTCDATETIME** 関数も用意されています。これは、PRINT ステートメントを利用して、次のように確認することができます。

```

PRINT SYSUTCDATETIME()      -- UTC (協定世界時)
PRINT SYSDATETIME()         -- タイムゾーンを含まない。datetime2 用
PRINT SYSDATETIMEOFFSET()   -- タイムゾーンを含む。    datetimeoffset 用

```

```

PRINT SYSUTCDATETIME()      -- UTC (協定世界時)
PRINT SYSDATETIME()         -- タイムゾーンを含まない。datetime2 用
PRINT SYSDATETIMEOFFSET()   -- タイムゾーンを含む。    datetimeoffset 用

```

2012-07-03 17:15:14.1787110
2012-07-04 02:15:14.1787110
2012-07-04 02:15:14.1787110 +09:00

Note : time、datetime2、datetimeoffset での桁数指定

time と datetime2、datetimeoffset データ型では、**time(7)** や **datetime2(7)** のように桁数を指定することもできます。既定値は (7) で、100 ナノ秒 (10^{-7}) 単位でデータを格納できます。**time(6)** と指定した場合は、マイクロ秒 (10 のマイナス 6 乗、 10^{-6}) 単位、**time(3)** と指定した場合は、ミリ秒 (10 のマイナス 3 乗、 10^{-3}) 単位でデータを格納できます。

日付時刻データの入力

次に、日付時刻データを手入力する場合を試してみましょう。日付時刻データは、次の書式で記述します。

```
datetime 型の場合      : 'YYYY/MM/DD hh:mm:ss.mmm'
datetime2 型の場合     : 'YYYY/MM/DD hh:mm:ss.nnnnnnn'
datetimeoffset 型の場合: 'YYYY/MM/DD hh:mm:ss.nnnnnnn {+|-}hh:mm'
```

日付と時刻の間は、半角スペースで区切り、時間と分、分と秒の間は「:」（コロン）、秒と 0.333 秒の間には「.」（ドット）を入れて区切ります。なお、日付の区切りの「/」（スラッシュ）は、代わりに「-」（ハイフン）を利用することもできます。

それでは、これを試してみましょう。

1. まずは、前の手順で作成した「**dateT**」テーブルへ次のように日付時刻を INSERT してみます。

```
INSERT INTO dateT
VALUES ( '2012/04/01 08:55:30.000'
        , '2012/04/01 08:55:30.000'
        , '2012/04/01 08:55:30.000'
        , '2012/04/01 08:55:30.000' )
SELECT * FROM dateT
```

	a	b	c	d
1	2012-07-04 02:11:52.617	2012-07-04 02:12:00	2012-07-04	02:11:52.6170000
2	2012-04-01 08:55:30.000	2012-04-01 08:56:00	2012-04-01	08:55:30.0000000

前の手順で確認したように datetime 型の「a」列は 0.333 秒単位、smalldatetime 型の「b」列は分単位、date 型の「c」列は日単位、time 型の「d」列は時間のみが格納されます。

時刻の省略時の動作

2. 次に、日付のみを指定して、時刻を省略してデータを追加してみましょう。

```
INSERT INTO dateT
VALUES ( '2012/05/01', '2012/05/01', '2012/05/01', '2012/05/01' )
SELECT * FROM dateT
```

```

INSERT INTO dateT
VALUES ( '2008/05/01', '2008/05/01', '2008/05/01', '2008/05/01' )

SELECT * FROM dateT

```

	a	b	c	d
1	2010-05-15 01:09:40.943	2010-05-15 01:10:00	2010-05-15	01:09:40.9430000
2	2008-04-01 08:55:30.000	2008-04-01 08:56:00	2008-04-01	08:55:30.0000000
3	2008-05-01 00:00:00.000	2008-05-01 00:00:00	2008-05-01	00:00:00.0000000

このように時刻を省略した場合には、00 時 00 分 00 秒 0000000 が補われます。

➡ 指定した日にちのデータのみを取得

3. 次に、WHERE 句の条件式へ日付のみを指定して、データを検索してみましょう。

```

SELECT * FROM dateT
WHERE a = '2012/04/01'

```

```

SELECT * FROM dateT
WHERE a = '2012/04/01'

```

	a	b	c	d
--	---	---	---	---

a 列のデータ型は datetime で、前の手順で「**2012/04/01 08:55:30.000**」データを追加していますが、「a='2012/04/01'」という条件ではヒットしません。時刻を省略した場合は、「a='2012/04/01 00:00:00.000'」と解釈されるからです。

したがって、datetime データ型の場合に、指定した日にちのデータを取得したい場合は、次のように記述する必要があります。

```

SELECT * FROM dateT
WHERE a >= '2012/04/01' AND a < '2012/04/02'

```

```

-- 2012/04/01 のデータを取得
SELECT * FROM dateT
WHERE a >= '2012/04/01' AND a < '2012/04/02'

```

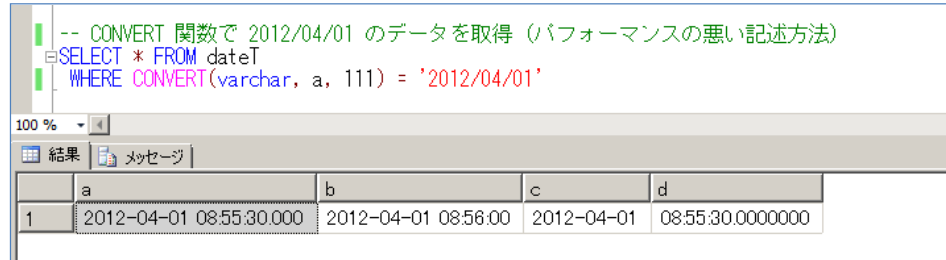
	a	b	c	d
1	2012-04-01 08:55:30.000	2012-04-01 08:56:00	2012-04-01	08:55:30.0000000

このように datetime 型へ時刻が格納されている場合には、2 つの条件式を指定しないと、指定した日のデータが取得できないことに注意しましょう。

Note : CONVERT や FORMAT 関数による日にち指定

詳しくは、次の STEP6 で説明しますが、**CONVERT** や **FORMAT** 関数を利用すると、**datetime** 型のデータを「YYYY/MM/DD」形式へ変換することができるので、指定した日にちのデータを取得したい場合は、次のように記述することもできます。

```
SELECT * FROM dateT
WHERE CONVERT(varchar, a, 111) = '2012/04/01'
```



このように **CONVERT** 関数の第 3 引数で「111」を指定すると、「YYYY/MM/DD」形式へ変換できるので、条件式を 1 つ書くだけで済みます。

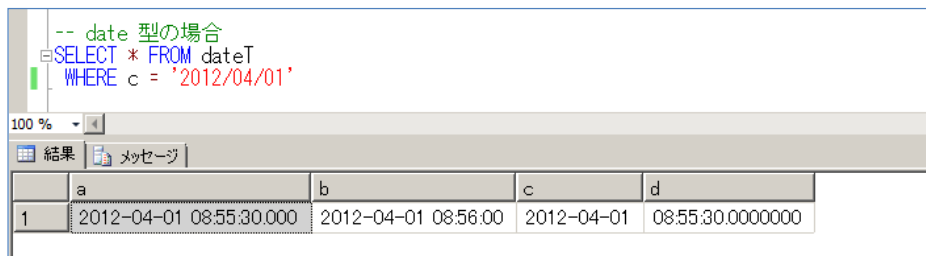
FORMAT 関数を利用する場合は、「**FORMAT(a, 'yyyy/MM/dd')**」と記述することで、「YYYY/MM/DD」形式へ変換することができます。

ただし、これらの関数を利用する方法は、内部的にはインデックスが利用されないテーブル スキャンまたはインデックス スキャン(全件走査)が行われるので、パフォーマンスの悪い記述方法です(筆者のお客様でも **CONVERT** 関数を利用してクエリを記述している方を多くみてきました)。したがって、前述の手順で試したように(冗長でも)2つの条件式を記述する方法を利用することをお勧めします。筆者のお客様では、2つの条件式へ変更したことで、80%以上ものパフォーマンス向上が実現したケースもあります。

また、後述の **date** データ型を利用すれば、時刻データは格納されないもので、「xx='2008/04/01'」という条件を記述しても検索にヒットし、パフォーマンスにも問題がありません。したがって、時刻を格納する必要のないデータの場合には、**date** データ型を積極的に利用することをお勧めします。

4. 次に、**date** データ型の「c」列に対して、**WHERE** 句の条件式で日付のみを指定してデータを検索してみましょう。

```
SELECT * FROM dateT
WHERE c = '2012/04/01'
```



date データ型には、時刻データが格納されませんので、条件式で、指定した日にちのデータを取得できるようになります。

STEP 6. 関数

SQL Server にはさまざまな関数が用意されています。この STEP では、関数を利用したデータの操作方法を説明します。

この STEP では、次のことを学習します。

- ✓ 日付と時刻に関する関数
- ✓ データ型の変換関数（CONVERT、CAST）
- ✓ 文字列操作の関数
- ✓ 数値操作の関数
- ✓ NULL 操作の関数
- ✓ ユーザー定義関数
- ✓ Oracle の関数との比較

6.1 日付と時刻に関する関数

➡ 日付と時刻に関する関数

SQL Server には、日付と時刻を操作するための関数として、主に次のものが用意されています。

関数	役割
GETDATE	現在の日付と時刻を取得 (datetime データ型対応) CURRENT_TIMESTAMPと記述しても可
SYSDATETIME	GETDATE の datetime2 データ型対応版
SYSDATETIMEOFFSET	GETDATE の datetimeoffset データ型対応版
YEAR	日付から年を取得 (結果は int 型)
MONTH	日付から月を取得 (結果は int 型)
DATEPART	日付/時刻から指定した部分を取得 (結果は int 型)
DATEADD	日付/時刻の加算を行う
DATEDIFF	日付/時刻の差を取得
EOMONTH	月末の取得
DATEFROMPARTS	文字列からの日付データの作成
FORMAT	日付データの書式の変更

➡ Let's Try

それでは、これを試してみましょう。ここでは、STEP4 で利用したのと同じ「**sampleDB**」データベース内の「**emp**」テーブル (以下のデータ) を利用して、関数を試してみましょう。

empno	empname	sal	hiredate	deptno
1	鈴木 一郎	500000	1998-04-01 00:00:00.000	10
2	小田 良夫	300000	1999-04-01 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	1996-04-01 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

➡ GETDATE で現在の日付と時刻を取得

1. まずは、前の STEP の復習も兼ねて、「**emp**」テーブルの「**hiredate**」(入社日) 列へ日付と時刻を指定して、データを INSERT してみます。

```
USE sampleDB
INSERT INTO emp
VALUES (11, 'xxx', 9999, '2006/04/01 08:55:30.000', 20)
SELECT * FROM emp WHERE empno = 11
```

```

INSERT INTO emp
VALUES (11, 'xxx', 9999, '2006/04/01 08:55:30.000', 20)

SELECT * FROM emp WHERE empno = 11

```

	empno	empname	sal	hiredate	deptno
1	11	xxx	9999	2006-04-01 08:55:30.000	20

2. 次に、**GETDATE** 関数を利用して「現在の日付と時刻」を取得し、それを「**emp**」テーブルの「**hiredate**」(入社日) 列へ格納してみます。

```

INSERT INTO emp
VALUES (12, 'yyy', 9999, GETDATE(), 20)

SELECT * FROM emp WHERE empno = 12

```

```

INSERT INTO emp
VALUES (12, 'yyy', 9999, GETDATE(), 20)

SELECT * FROM emp WHERE empno = 12

```

	empno	empname	sal	hiredate	deptno
1	12	yyy	9999	2012-07-04 02:27:10.363	20

現在の日付と時刻が、「hiredate」列へ格納されていることを確認できます。

➡ YEAR 関数で年のみを取得

1. 次に、**YEAR** 関数を利用して、「hiredate」(入社日) 列の“年だけ”を取得してみます。

```
SELECT YEAR(hiredate), * FROM emp
```

(列名なし)	empno	empname	sal	hiredate	deptno
1998	1	鈴木 一郎	500000	1998-04-01 00:00:00.000	10
1999	2	小田 良夫	300000	1999-04-01 00:00:00.000	10
NULL	3	浅田 ゆかり	NULL	NULL	20
1985	4	田村 健一	700000	1985-04-01 00:00:00.000	10
1996	5	中野 浩之	500000	1996-04-01 00:00:00.000	10
1997	6	内藤 太郎	500000	1997-04-01 00:00:00.000	10
2000	7	大和田 聡	250000	2000-04-01 00:00:00.000	20
2006	8	長谷川 忍	300000	2006-04-01 00:00:00.000	20
2006	9	Geof Cruise	600000	2006-04-01 00:00:00.000	20
2006	11	xxx	9999	2006-04-01 08:55:30.000	20
2012	12	yyy	9999	2012-07-04 02:27:10.363	20

YEAR(hiredate) によって
入社日のうちの“年”を取得

このように YEAR 関数を利用すると、「1998」や「1999」など年だけを取得できるようになります。なお、取得した年は、int (4 バイト整数) 型です。

➡ MONTH 関数で月のみを取得

- 次に、**MONTH** 関数を利用して、「hiredate」（入社日）列の“月だけ”を取得してみます。

```
SELECT MONTH(hiredate), * FROM emp
```

(列名なし)	empno	empname	sal	hiredate	deptno
4	1	鈴木 一郎	500000	1998-04-01 00:00:00.000	10
4	2	小田 良夫	300000	1999-04-01 00:00:00.000	10
NULL	3	浅田 ゆかり	NULL	NULL	20
4	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
4	6	内藤 太郎	500000	1997-04-01 00:00:00.000	10
4	7	大和田 聡	250000	2000-04-01 00:00:00.000	20
4	8	長谷川 忍	300000	2006-04-01 00:00:00.000	20
4	9	Geof Cruise	800000	2006-04-01 00:00:00.000	20
4	11	xxx	9999	2006-04-01 08:55:30.000	20
7	12	yyy	9999	2012-07-04 02:27:10.363	20

MONTH(hiredate) によって
入社日のうちの“月”を取得

このように MONTH 関数を利用すると、月だけを取得できるようになります。また、YEAR 関数と同様、取得した値は int 型です。

➡ DATEPART による日付と時刻の部分取得

YEAR と MONTH 関数は、年と月の取得でしたが、**DATEPART** 関数を利用すると、日付と時刻の任意の一部分を取得できるようになります。Part は「部分」「一部」という意味です。構文は、次のとおりです。

```
DATEPART ( datepart, 日付 )
```

第 1 引数の datepart を、「year」と指定すれば“年”を取得でき、YEAR 関数と同じ結果を取得できます。また、「month」とすれば“月”で MONTH 関数と同じ結果、「day」と指定すれば“日”、「hour」と指定すれば“時間”、「minute」と指定すれば“分”を取得できるようになります。

それでは、これを試してみましょう。

- DATEPART** 関数を利用して、「emp」テーブルの「hiredate」から“日”と“時間”を取り出してみましょう。

```
SELECT DATEPART(day, hiredate), DATEPART(hour, hiredate), *  
FROM emp
```

(列名なし)	(列名なし)	empno	empname	sal	hiredate	deptno
1	0	1	鈴木 一郎	500000	1998-04-01 00:00:00.000	10
1	0	2	小田 良夫	300000	1999-04-01 00:00:00.000	10
NULL	NULL	3	浅田 ゆかり	NULL	NULL	20
1	0	4	田村 健一	700000	1985-04-01 00:00:00.000	10
1	0	5	中野 浩之	500000	1996-04-01 00:00:00.000	10
1	0	6	内藤 太郎	500000	1997-04-01 00:00:00.000	10
1	0	7	大和田 聡	250000	2000-04-01 00:00:00.000	20
1	0	8	長谷川 忍	300000	2006-04-01 00:00:00.000	20
1	0	9	Geof Cruise	600000	2006-04-01 00:00:00.000	20
1	8	11	xxx	9999	2006-04-01 08:55:30.000	20
4	2	12	yyy	9999	2012-07-04 02:27:10.363	20

DATEPART(day, hiredate)

DATEPART(hour, hiredate)

このように DATEPART 関数を利用すると、日付と時刻の任意の一部分を取得できるようになります。

Note : DATEPART 関数で指定できるそのほかの datepart

DATEPART 関数の第 1 引数 (datepart) には、ほかにも指定できるものがあります。例えば、「dayofyear」で“その年の 1 月 1 日から数えた日にち”、「week」で“その年の何週目か”などを取得できます。これらの詳細は、SQL Server 2012 のオンライン ブック (Books Online) の以下のトピックへ詳しく記載されています。

「SQL Server データベース エンジン」→「Transact-SQL リファレンス」
→「組み込み関数」→「日付と時刻のデータ型および関数」→「DATEPART」

Microsoft ヘルプ ビュアー 1.1 - カタログ SQLSERVER_110_JA-JP

SQL Server データベース エンジン
 新機能 (データベース エンジン)
 SQL Server データベース エンジンの日バージョンとの互換性
 SQL Server 管理ツールの日バージョンとの互換性
 データベース エンジンの機能とタスク
 テクニカルリファレンス (データベース エンジン)
 Transact-SQL リファレンス (データベース エンジン)
 予約済みキーワード (Transact-SQL)
 Transact-SQL 構文表記規則 (Transact-SQL)
 BACKUP ステートメントと RESTORE ステートメント (Transact-SQL)
 組み込み関数 (Transact-SQL)
 集計関数 (Transact-SQL)
 分析関数 (Transact-SQL)
 照合順序関数 (Transact-SQL)
 構文関数 (Transact-SQL)
 変換関数 (Transact-SQL)
 暗号化関数 (Transact-SQL)
 カーソル関数 (Transact-SQL)
 データ型の関数 (Transact-SQL)
 日付と時刻のデータ型および関数 (Transact-SQL)
 @@DATEFIRST (Transact-SQL)
 CURRENT_TIMESTAMP (Transact-SQL)
 DATEADD (Transact-SQL)
 DATEDIFF (Transact-SQL)
 DATEFROMPARTS (Transact-SQL)
 DATENAME (Transact-SQL)
DATEPART (Transact-SQL)
 DATETIME2FROMPARTS (Transact-SQL)
 DATETIMEFROMPARTS (Transact-SQL)
 DATETIMEOFFSETFROMPARTS (Transact-SQL)

DATEPART (Transact-SQL) ×

次の表は、SELECT DATEPART(datepart, '2007-10-30 12:15:32.1234567 +05:10') で、すべての datepart 引数と、対応する戻り値を一覧にしたものです。
 date 引数のデータ型は **datetimeoffset(7)** です。
 datepart に **nanosecond** を指定した場合の戻り値の精度は 9 桁 (123456700) で、最後の 2 桁は 00 です。

datepart	戻り値
year, yyyy, yy	2007
quarter, qq, q	4

➡ DATEADD による日付の加算と減算

DATEADD 関数を利用すると、日付と時刻の任意の部分で加算と減算を行えるようになります。構文は、次のとおりです。

DATEADD (datepart, 数値, 日付)

第 1 引数の datepart は、DATEPART 関数のときと同じように、「year」や「month」など加算や減算の対象にしたいものを指定します。第 2 引数の数値は、第 3 引数に対して加算または減算したい値を指定します。減算の場合は、マイナス付きの数値を記述します。

それでは、これを試してみましょう。

1. まずは、「emp」テーブルの「hiredate」（入社日）列へ「10 年加算」した値を取得してみます。

```
SELECT DATEADD(year, 10, hiredate), * FROM emp
```

(列名なし)	empno	empname	sal	hiredate	deptno
2008-04-01 00:00:00.000	1	鈴木 一郎	500000	1998-04-01 00:00:00.000	10
2009-04-01 00:00:00.000	2	小田 良夫	300000	1999-04-01 00:00:00.000	10
NULL	3	浅田 ゆかり	NULL	NULL	20
1995-04-01 00:00:00.000	4	田村 健一	700000	1985-04-01 00:00:00.000	10
2006-04-01 00:00:00.000	5	中野 浩之	500000	1996-04-01 00:00:00.000	10
2007-04-01 00:00:00.000	6	内藤 太郎	500000	1997-04-01 00:00:00.000	10
2010-04-01 00:00:00.000	7	大和田 聡	250000	2000-04-01 00:00:00.000	20
2016-04-01 00:00:00.000	8	長谷川 忍	300000	2006-04-01 00:00:00.000	20
2016-04-01 00:00:00.000	9	Geof Cruise	600000	2006-04-01 00:00:00.000	20
2016-04-01 08:55:30.000	11	xxx	9999	2006-04-01 08:55:30.000	20
2022-07-04 02:27:10.363	12	yyy	9999	2012-07-04 02:27:10.363	20

DATEADD(yaer, 10, hiredate)
で入社日の“年”に +10 加算

第 1 引数へ“year” 第 2 引数へ“10”を指定することで、hiredate（入社日）の“年”に対して“+10”した結果を取得することができます。

2. 次に、DATEADD 関数を WHERE 句で利用してみましょう。次のように記述して、“現在、入社してから 10 年以上経っている社員”を取得してみます。

```
SELECT * FROM emp
WHERE DATEADD(year, 10, hiredate) < GETDATE()
```

	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
6	7	大和田 聡	250000	2000-04-01 00:00:00.000	20

このように関数は、WHERE 句の条件式で利用することもできます。

➡ EOMONTH による月末の取得

EOMONTH 関数は、SQL Server 2012 からサポートされた関数で、月末（月の最後の日付）を取得することができます。構文は、次のとおりです。

```
EOMONTH(日付)
```

それでは、これを試してみましょう。

1. 「emp」テーブルの「hiredate」（入社日）列の月末を取得してみます。

```
SELECT EOMONTH(hiredate), hiredate FROM emp
```

	(列名なし)	hiredate
1	1998-04-30	1998-04-01 00:00:00.000
2	1999-04-30	1999-04-01 00:00:00.000
3	NULL	NULL
4	1985-04-30	1985-04-01 00:00:00.000
5	1996-04-30	1996-04-01 00:00:00.000
6	1997-04-30	1997-04-01 00:00:00.000
7	2000-04-30	2000-04-01 00:00:00.000
8	2006-04-30	2006-04-01 00:00:00.000
9	2006-04-30	2006-04-01 00:00:00.000

「emp」テーブルの「hiredate」列の月末を取得できたことを確認できます。

➡ DATEFROMPARTS で文字列から日付データを作成

DATEFROMPARTS 関数は、SQL Server 2012 からサポートされた関数で、文字列から日付データを作成することができます。構文は、次のとおりです。

```
DATEFROMPARTS( '年', '月', '日' )
```

第 1 引数には年となる値、第 2 引数には月となる値、第 3 引数には日となる値をそれぞれ記述します。

それでは、これを試してみましょう。

1. 「2012」と「12」、「12」という文字列から日付を作成してみます。

```
SELECT DATEFROMPARTS( '2012', '12', '12' )
```

(列名なし)
2012-12-12

「2012」と「12」、「12」から「2012/12/12」（date 型の日付）を作成できたことを確認できます。DATEFROMPARTS は、Parts（パーツ）From（から）date 型を作成するという意味です。他の日付データ型を作成できる関数も用意されていて、datetime 型の「**DATETIMEFROMPARTS**」、smalldatetime 型の「**SMALLDATETIMEFROMPARTS**」、datetime2 型の「**DATETIME2FROMPARTS**」、datetimeoffset 型の「**DATETIMEOFFSETFROMPARTS**」などがあります。

➡ FORMAT で日付データの書式を変更

FORMAT 関数も、SQL Server 2012 からサポートされた関数で、日付データの書式を変更することができます。構文は、次のとおりです。

FORMAT (日付, 書式)

第 1 引数には日付となる値、第 2 引数には変更後の書式を指定します。

それでは、これを試してみましょう。

1. 「emp」テーブルの「hiredate」列の日付データを「XXXX 年 XX 月 XX 日」という形で取得してみます。

```
SELECT FORMAT(hiredate, 'yyyy年MM月dd日'), hiredate FROM emp
```

	(列名なし)	hiredate
1	1998年04月01日	1998-04-01 00:00:00.000
2	1999年04月01日	1999-04-01 00:00:00.000
3	NULL	NULL
4	1985年04月01日	1985-04-01 00:00:00.000
5	1996年04月01日	1996-04-01 00:00:00.000
6	1997年04月01日	1997-04-01 00:00:00.000
7	2000年04月01日	2000-04-01 00:00:00.000
8	2006年04月01日	2006-04-01 00:00:00.000
9	2006年04月01日	2006-04-01 00:00:00.000

「hiredate」列の日付データを「1998 年 04 月 01 日」のような形で取得できたことを確認できます。**yyyy** は 4 桁の年、**MM** は 2 桁の月、**dd** は 2 桁の日を取得するための .NET Framework の**書式指定子**（書式を変更するためのキーワード）です。FORMAT 関数では、.NET Framework でサポートされる書式指定子を利用してデータの書式を変更することができます。

6.2 データ型の変換関数： CONVERT、CAST

➡ CONVERT と CAST

CONVERT と **CAST** 関数は、データ型を変換するための関数です。SQL Server は、Visual Basic 系の言語とは違って、データ型があいまいではなく、型に厳しい言語なので、数値と文字列を連結する場合などは、データ型の変換エラーが発生します（型変換が必要になります）。

CONVERT と CAST 関数は、次のように利用します。

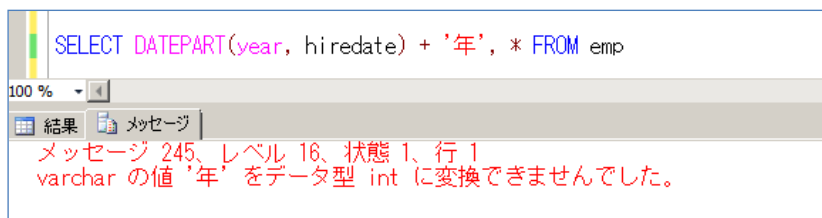
CONVERT (変換後のデータ型, 変換したいデータ [, 日付書式のオプション])
CAST (変換したいデータ AS 変換後のデータ型)

➡ Let's Try

それでは、これを試してみましょう。

1. まずは、「**emp**」テーブルの「**hiredate**」（入社日）列から “年” のみを取得し、このデータへ “年” という文字列を連結して、「1998 年」や「2006 年」のように表示してみます。

```
SELECT DATEPART(year, hiredate) + '年', * FROM emp
```



結果はエラーになります。DATEPART の結果（戻り値）は、**int** データ型なので、“年” という文字列（char データ型）と連結しようとするためエラーが発生するためです（関数の戻り値のデータ型は、SQL Server オンライン ブックの「Transact-SQL リファレンス」内の各関数のヘルプへ記載されています）。このエラーを回避するには、CONVERT または CAST 関数を利用して、数値データを文字列へ変換する必要があります。

➡ CONVERT 関数

2. 次に、**CONVERT** 関数を利用して、DATEPART 関数で取得した「**hiredate**」（入社日）の年を **char(4)** データ型へ変換してみます。

```
SELECT CONVERT( char(4), DATEPART(year, hiredate) ) + '年', *  
FROM emp
```

```
SELECT CONVERT( char(4), DATEPART(year, hiredate) ) + '年', *
FROM emp
```

	(列名なし)	empno	empname	sal	hiredate	deptno
1	1998年	1	鈴木 一郎	500000	1998-04-01 00:00:00.000	10
2	1999年	2	小田 良夫	300000	1999-04-01 00:00:00.000	10
3	NULL	3	浅田 ゆかり	NULL	NULL	20
4	1985年	4	田村 健一	700000	1985-04-01 00:00:00.000	10
5	1996年	5	中野 浩之	500000	1996-04-01 00:00:00.000	10
6	1997年	6	内藤 太郎	500000	1997-04-01 00:00:00.000	10
7	2000年	7	大和田 聡	250000	2000-04-01 00:00:00.000	20
8	2006年	8	長谷川 忍	300000	2006-04-01 00:00:00.000	20
9	2006年	9	Geof Cruise	600000	2006-04-01 00:00:00.000	20
10	2006年	11	xxx	9999	2006-04-01 08:55:30.000	20
11	2012年	12	yyy	9999	2012-07-04 02:27:10.363	20

今度はエラーにはならず、正しく文字列が連結されていることを確認できます。

なお、**FORMAT** 関数を利用する場合は、「**FORMAT(hiredate, 'yyyy 年')**」と記述することで同じ結果を取得することができます（**FORMAT** 関数の戻り値は **nvarchar** のため）。

➡ CAST 関数

- 次に、**CONVERT** 関数の代わりに、**CAST** 関数を利用してみましょう。

```
SELECT CAST( DATEPART(year, hiredate) AS char(4) ) + '年', * FROM emp
```

```
SELECT CAST( DATEPART(year, hiredate) AS char(4) ) + '年', * FROM emp
```

	(列名なし)	empno	empname	sal	hiredate	deptno
1	1998年	1	鈴木 一郎	500000	1998-04-01 00:00:00.000	10
2	1999年	2	小田 良夫	300000	1999-04-01 00:00:00.000	10
3	NULL	3	浅田 ゆかり	NULL	NULL	20
4	1985年	4	田村 健一	700000	1985-04-01 00:00:00.000	10
5	1996年	5	中野 浩之	500000	1996-04-01 00:00:00.000	10
6	1997年	6	内藤 太郎	500000	1997-04-01 00:00:00.000	10
7	2000年	7	大和田 聡	250000	2000-04-01 00:00:00.000	20
8	2006年	8	長谷川 忍	300000	2006-04-01 00:00:00.000	20
9	2006年	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。**CAST** 関数では、引数が 1 つのみで、変換したいデータの後に「**AS**」を記述することに注意してください。

Note : CONVERT と CAST の使い分け

CAST は ANSI SQL92 規格、**CONVERT** は SQL Server 独自の関数なので、**CAST** を利用したほうが他のデータベース製品を利用する場合に役立ちます。ただ、SQL Server 6.5 までは **CAST** がサポートされていなかったため、古くからの SQL Server ユーザーは **CONVERT** 関数を好む傾向があります。もちろん、どちらを利用しても問題ありません。

➡ CONVERT による日付書式の変換（文字列への変換）

次に、CONVERT 関数を利用して、日付と時刻を文字列へ変換してみましょう。

4. まずは、「emp」テーブルの「hiredate」（入社日）列を **varchar** データ型へ変換してみます。

```
SELECT CONVERT( varchar, hiredate ), * FROM emp
```

	(列名なし)	empno	empname	sal	hiredate	deptno
1	04 1 1998 12:00AM	1	鈴木 一郎	500000	1998-04-01 00:00:00.000	10
2	04 1 1999 12:00AM	2	小田 良夫	300000	1999-04-01 00:00:00.000	10
3	NULL	3	浅田 ゆかり	NULL	NULL	20
4	04 1 1985 12:00AM	4	田村 健一	700000	1985-04-01 00:00:00.000	10
5	04 1 1996 12:00AM	5	中野 浩之	500000	1996-04-01 00:00:00.000	10
6	04 1 1997 12:00AM	6	内藤 太郎	500000	1997-04-01 00:00:00.000	10
7	04 1 2000 12:00AM	7	大和田 聡	250000	2000-04-01 00:00:00.000	20
8	04 1 2006 12:00AM	8	長谷川 忍	300000	2006-04-01 00:00:00.000	20
9	04 1 2006 12:00AM	9	Geof Cruise	600000	2006-04-01 00:00:00.000	20
10	04 1 2006 8:55AM	11	xxx	9999	2006-04-01 08:55:30.000	20
11	07 4 2012 2:27AM	12	yyy	9999	2012-07-04 02:27:10.363	20

変換先のデータ型には、“varchar” とだけ指定し、最大バイト数を指定していませんが、この場合は変換元の最大サイズに合わせて自動調整してくれます。これで日付を文字列へ変換することができます。しかし、結果は「月 日 年 時間」（MM DD YYYY）の順に表示されています。これを「年/月/日」（YYYY/MM/DD）の順で表示されるようにするには、CONVERT の第3引数で「111」または「11」を指定するようにします。

5. それでは、CONVERT 関数の第3引数へ「111」を指定して試してみましょう。

```
SELECT CONVERT( varchar, hiredate, 111 ), * FROM emp
```

	(列名なし)	empno	empname	sal	hiredate	deptno
1	1998/04/01	1	鈴木 一郎	500000	1998-04-01 00:00:00.000	10
2	1999/04/01	2	小田 良夫	300000	1999-04-01 00:00:00.000	10
3	NULL	3	浅田 ゆかり	NULL	NULL	20
4	1985/04/01	4	田村 健一	700000	1985-04-01 00:00:00.000	10
5	1996/04/01	5	中野 浩之	500000	1996-04-01 00:00:00.000	10
6	1997/04/01	6	内藤 太郎	500000	1997-04-01 00:00:00.000	10
7	2000/04/01	7	大和田 聡	250000	2000-04-01 00:00:00.000	20
8	2006/04/01	8	長谷川 忍	300000	2006-04-01 00:00:00.000	20
9	2006/04/01	9	Geof Cruise	600000	2006-04-01 00:00:00.000	20
10	2006/04/01	11	xxx	9999	2006-04-01 08:55:30.000	20
11	2012/07/04	12	yyy	9999	2012-07-04 02:27:10.363	20

結果は、**YYYY/MM/DD** 形式で表示されていることを確認できます。このように CONVERT 関数の第3引数では、各国の日付／時刻表示に合わせた変換が行えるようになっています。

なお、**FORMAT** 関数を利用する場合は、「**FORMAT(hiredate, 'yyyy/MM/dd')**」と記述することで同じ結果を取得することができます。

6. 次に、**CONVERT** 関数の第 3 引数へ「**114**」を指定して、実行してみます。

```
SELECT CONVERT( varchar, hiredate, 114 ), * FROM emp
```

	(列名なし)	empno	empname	sal	hiredate	deptno
1	00:00:00.000	1	鈴木 一郎	500000	1998-04-01 00:00:00.000	10
2	00:00:00.000	2	小田 良夫	300000	1999-04-01 00:00:00.000	10
3	NULL	3	浅田 ゆかり	NULL	NULL	20
4	00:00:00.000	4	田村 健一	700000	1985-04-01 00:00:00.000	10
5	00:00:00.000	5	中野 浩之	500000	1996-04-01 00:00:00.000	10
6	00:00:00.000	6	内藤 太郎	500000	1997-04-01 00:00:00.000	10
7	00:00:00.000	7	大和田 聡	250000	2000-04-01 00:00:00.000	20
8	00:00:00.000	8	長谷川 忍	300000	2006-04-01 00:00:00.000	20
9	00:00:00.000	9	Geof Cruise	600000	2006-04-01 00:00:00.000	20
10	08:55:30.000	11	xxx	9999	2006-04-01 08:55:30.000	20
11	02:27:10.363	12	yyy	9999	2012-07-04 02:27:10.363	20

結果は、時刻のみを取得できていることを確認できます。**CONVERT** 関数の第 3 引数では、「**114**」または「**14**」を指定すると、時刻のみを取得できるようになります。

なお、**FORMAT** 関数を利用する場合は、「**FORMAT(hiredate, 'HH:mm:ss.fff')**」と記述することで同じ結果を取得することができます。

Note : **CONVERT** の第 3 引数で指定できる値

CONVERT 関数の第 3 引数で指定できる値は、SQL Server オンライン ブックの「**SQL Server データベース エンジン**」→「**Transact-SQL リファレンス**」→「**組み込み関数**」→「**変換関数**」→「**CAST および CONVERT**」トピックに記載されています。このトピックへは、クエリ エディターで **CONVERT** 関数を選択した状態で **[F1]** キーを押下すると、自動的にオンライン ブックが開いて、このトピックへジャンプすることもできます。

Style Code	Language	Format
10	米国	mm-dd-yy
11	日本	yy/mm/dd
12	ISO	yyymmdd
13 または 113 (1, 2)	ヨーロッパの既定値 + ミリ秒	dd mon yyyy hh:mm:ss:mmmm(24h)

Note : int データ型に対する数値演算の結果が int データ型になることに注意

int データ型に格納したデータに対して、数値演算（除算や集計関数で計算するなど）を行うと、計算結果も int データ型なることに注意する必要があります。たとえば、集計関数の AVG（平均を取得できる関数）を利用して、平均を計算したとしても、結果は int データ型になってしまいます。これでは、計算結果に小数点以下の値があった場合に、その値は切り捨てられてしまいます。

これは、「emp」テーブルの「sal」（給与）列を利用して試すことができます。この列に対して、AVG 関数を利用して、全社員の平均給与を計算してみます（AVG 関数については、本自習書シリーズの「SQL 基礎の基礎」で説明しています）。

```
SELECT AVG(sal) FROM emp
```

	(列名なし)
1	366999

計算結果は「366999」となり、小数点以下の値が切り捨てられています。小数点以下の値が切り捨てられないようにするには、次のように CONVERT 関数を利用して、計算対象となる値を float や decimal データ型など、小数点以下を扱えるデータ型へ変換するようにします。

```
SELECT AVG( CONVERT( float, sal) ) FROM emp
```

	(列名なし)
1	366999.8

計算結果は「366999.8」となり、小数点以下の値を取得できたことを確認できます。

なお、次のように AVG 関数による計算結果に対して、float 型へ変換する場合は、小数点以下を取得することはできません。

```
SELECT CONVERT( float, AVG(sal) ) FROM emp
```

	(列名なし)
1	366999

小数点以下の値を取得したい場合は、あくまでも計算前に（計算対象となるデータを）変換しておくことに注意してください。

6.3 文字列操作の関数

文字列操作の関数

文字列操作が行える関数には、主に次のものが用意されています。

関数	役割
UPPER、LOWER	大文字、小文字変換
RTRIM/LTRIM	右/左から半角スペースの削除
REPLACE STUFF	文字列の置換
RIGHT/LEFT	右/左から部分抽出
SUBSTRING	部分抽出
LEN	文字列の長さ
DATALLENGTH	文字列のバイト数
CHARINDEX	文字列内の検索
ASCII	文字の ASCII コード取得
CHAR	ASCII コードを文字変換

大文字と小文字変換が行える「**UPPER**」と「**LOWER**」関数、半角スペースをとる「**RTRIM**」関数、文字を部分抽出する「**SUBSTRING**」関数などがよく利用する関数になります。

Let's Try

それでは、これらの関数を試してみましょう。

- まずは、**UPPER** 関数を利用して、「emp」テーブルの「empname」列の英字データを大文字へ変換してみましょう。

```
SELECT UPPER(empname), * FROM emp
```

The screenshot shows a SQL query window with the query: `SELECT UPPER(empname), * FROM emp`. Below the query, the results are displayed in a table grid. The first column is labeled '(列名なし)' and contains the uppercase names. The other columns are empno, empname, sal, hiredate, and deptno. The row for 'Geof Cruise' is highlighted in red.

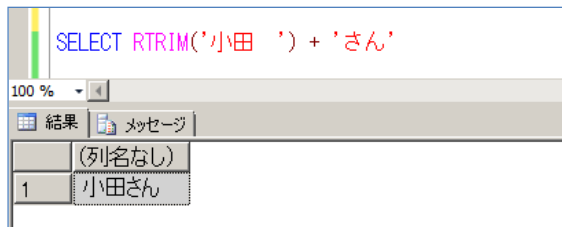
	(列名なし)	empno	empname	sal	hiredate	deptno
1	鈴木 一郎	1	鈴木 一郎	500000	1998-04-01 00:00:00.000	10
2	小田 良夫	2	小田 良夫	300000	1999-04-01 00:00:00.000	10
3	浅田 ゆかり	3	浅田 ゆかり	NULL	NULL	20
4	田村 健一	4	田村 健一	700000	1985-04-01 00:00:00.000	10
5	中野 浩之	5	中野 浩之	500000	1996-04-01 00:00:00.000	10
6	内藤 太郎	6	内藤 太郎	500000	1997-04-01 00:00:00.000	10
7	大和田 聡	7	大和田 聡	250000	2000-04-01 00:00:00.000	20
8	長谷川 忍	8	長谷川 忍	300000	2006-04-01 00:00:00.000	20
9	GEOF CRUISE	9	Geof Cruise	600000	2006-04-01 00:00:00.000	20
10	XXX	11	xxx	9999	2006-04-01 08:55:30.000	20
11	YYY	12	yyy	9999	2012-07-04 02:27:10.363	20

Geof Cruise さんのデータを大文字へ変換して取得できていることを確認できます。

◆ RTRIM、LTRIM

1. 続いて、次のように文字列の右側へ半角スペースを 3 つ付加した文字列 "小田 " というデータを **RTRIM** 関数を利用して、"さん" という文字列と連結してみましょう。

```
SELECT RTRIM('小田 ') + 'さん'
```



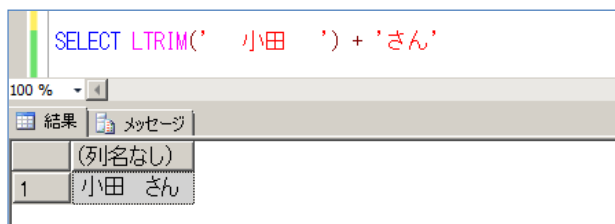
RTRIM 関数によって、文字列の右側の半角スペースを削除できたことを確認できます。RTRIM 関数は、Right Trim の略で、文字列の右側 (Right) の半角スペースを削除できる関数です。Trim は「取り除く」「切り取る」という意味です。

Note : SQL Server では FROM 句の省略が可能

SQL Server では、関数の効果を試すために、この例のように SELECT ステートメントの FROM 句を省略して利用することができます (関数の利用方法を調べる場合に便利です)。ちなみに、Oracle では、このような使い方はできず、FROM 句が必須になります。Oracle で関数の効果を試したい場合には、擬似テーブルの「DUAL」を利用して「SELECT 関数 FROM DUAL」のように記述する必要があります。

2. 続いて、次のように文字列の左右に半角スペースを 3 つ付加した文字列 " 小田 " というデータを **LTRIM** 関数を利用して、"さん" という文字列と連結してみましょう。

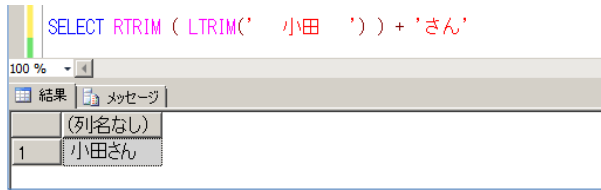
```
SELECT LTRIM(' 小田 ') + 'さん'
```



LTRIM 関数によって、文字列の左側の半角スペースを削除できたことを確認できます。LTRIM 関数は、文字列の左側 (Left) の半角スペースを削除できる関数です。

3. 次に、RTRIM 関数と LTRIM 関数を入れ子にして、左右の両方の半角スペースを削除 (左右同時に削除する) してみましょう。

```
SELECT RTRIM ( LTRIM(' 小田 ') ) + 'さん'
```

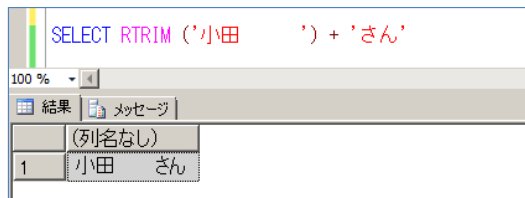



Oracle では、左右両方の半角スペースを取り除く関数として「TRIM」が用意されていますが、SQL Server には用意されていないので、このように RTRIM と LTRIM を入れ子にして利用する必要があります。

Note : REPLACE で全角スペースを削除

RTRIM と LTRIM では、“全角” のスペースを削除することができないことに注意しましょう。たとえば、次のように “山田 ” という文字列（右側に 3 つの全角スペースを付加した文字列）の場合は、スペースを削除することができません。

```
SELECT RTRIM ( '小田 ' ) + 'さん'
```



この場合は、**REPLACE** という関数を使って、全角スペースを削除するようにします。REPLACE は、文字列内の一部を、別の文字列へ置換できる関数なので、全角スペースを空の文字列へ置換するようにすれば、全角スペースを削除することができます。

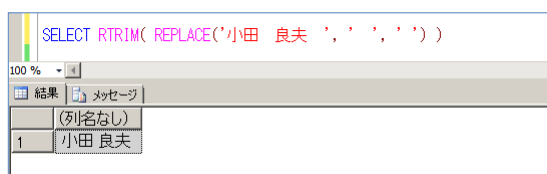
```
SELECT REPLACE ( '小田 ', ' ', '' ) + 'さん'
```



REPLACE 関数では、第 2 引数へ置換対象となる文字列（全角スペース）を指定し、第 3 引数へ置換後の文字列（空の文字列）を指定します。

ただし、上の例では “小田 良夫 ” という文字列（姓と名の間が全角スペース）だった場合に、姓と名の間の全角スペースも一緒に削除してしまいます。これを削除しないようにするには、次のように REPLACE 関数で全角スペースを一度半角スペースへ変換し、その後 RTRIM または LTRIM 関数を利用して、左右の半角スペースを削除するようにします。

```
SELECT RTRIM ( REPLACE ( '小田 良夫 ', ' ', ' ' ) )
```



REPLACE 関数の第 3 引数を、空の文字列から半角スペースへ変換しているところがポイントです。

➡ RIGHT と LEFT、SUBSTRING による部分抽出

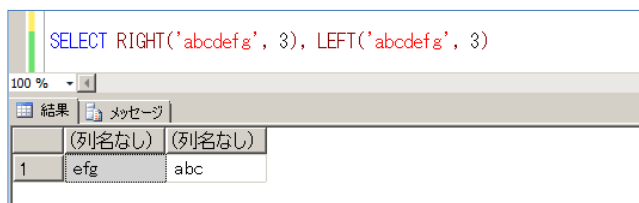
RIGHT 関数は、文字列を右側から n 文字分、**LEFT** 関数は、左側から n 文字分を取り出すことができる関数です。また、**SUBSTRING** 関数は、任意の場所から文字列を n 文字分取り出すことができる関数です。これらは、次のように利用します。

```
RIGHT ( ' 文字列' , 取り出したい文字数)
LEFT ( ' 文字列' , 取り出したい文字数)
SUBSTRING ( ' 文字列' , 開始位置, 取り出したい文字数)
```

それでは、これを試してみましょう。

1. まずは、**RIGHT** 関数と **LEFT** 関数を利用して “abcdefg” という文字列に対して、右または左から 3 文字分を取り出してみます。

```
SELECT RIGHT('abcdefg', 3), LEFT('abcdefg', 3)
```



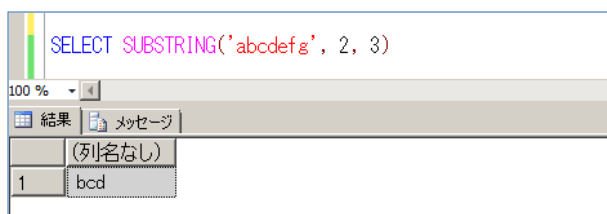
The screenshot shows a SQL query window with the text `SELECT RIGHT('abcdefg', 3), LEFT('abcdefg', 3)`. Below the query window, the 'Results' pane displays a table with two columns, both labeled '(列名なし)'. The first row contains the values 'efg' and 'abc'.

	(列名なし)	(列名なし)
1	efg	abc

このように、**RIGHT** と **LEFT** では、第 2 引数へ取り出したい（部分抽出したい）文字数を指定します。“3” と指定した場合は、右または左から 3 文字分を取り出すことができます。

2. 続いて、**SUBSTRING** 関数を利用して、文字列の 2 文字目から 3 文字分を取り出してみましょう。

```
SELECT SUBSTRING('abcdefg', 2, 3)
```



The screenshot shows a SQL query window with the text `SELECT SUBSTRING('abcdefg', 2, 3)`. Below the query window, the 'Results' pane displays a table with one column labeled '(列名なし)'. The first row contains the value 'bcd'.

	(列名なし)
1	bcd

SUBSTRING 関数では、第 2 引数へ何文字目から取り出したいかを指定し、第 3 引数へ取り出したい文字数を指定します。この例では、“2”と “3” を指定しているので、2 文字目から 3 文字分を取り出すことができます。

◆ そのほかの文字列操作関数

SQL Server 2012 には、上記で紹介したもの以外にも、多くの文字列操作関数が用意されています。これらは、SQL Server オンライン ブックの「**Transact-SQL リファレンス**」→「**組み込み関数**」→「**文字列関数**」トピックへ記載されています。

Microsoft ヘルプ ビューアー 1.1 - カタログ SQLSERVER_110_JA-JP

SQL Server データベース エンジン
 新機能 (データベース エンジン)
 SQL Server データベース エンジン の旧バージョンとの互換性
 SQL Server 管理ツールの旧バージョンとの互換性
 データベース エンジンの機能とタスク
 テクニカルリファレンス (データベース エンジン)
 Transact-SQL リファレンス (データベース エンジン)
 予約済みキーワード (Transact-SQL)
 Transact-SQL 構文表記規則 (Transact-SQL)
 BACKUP ステートメントと RESTORE ステートメント (Transact-SQL)
 組み込み関数 (Transact-SQL)
 集計関数 (Transact-SQL)
 分析関数 (Transact-SQL)
 照合順序関数 (Transact-SQL)
 構成関数 (Transact-SQL)
 変換関数 (Transact-SQL)
 暗号化関数 (Transact-SQL)
 カーソル関数 (Transact-SQL)
 データ型の関数 (Transact-SQL)
 日付と時刻のデータ型および関数 (Transact-SQL)
 論理関数 (Transact-SQL)
 数値関数 (Transact-SQL)
 メタデータ関数 (Transact-SQL)
 ODBC スカラー関数 (Transact-SQL)
 順位付け関数 (Transact-SQL)
 レプリケーション関数 (Transact-SQL)
 行セット関数 (Transact-SQL)
 セキュリティ関数 (Transact-SQL)
文字列関数 (Transact-SQL)
 ASCII (Transact-SQL)
 CHAR (Transact-SQL)
 CHARINDEX (Transact-SQL)
 CONCAT (Transact-SQL)
 DIFFERENCE (Transact-SQL)
 FORMAT (Transact-SQL)
 LEFT (Transact-SQL)
 LEN (Transact-SQL)
 LOWER (Transact-SQL)
 LTRIM (Transact-SQL)
 NCHAR (Transact-SQL)
 PATINDEX (Transact-SQL)
 QUOTENAME (Transact-SQL)
 REPLACE (Transact-SQL)
 REPLICATE (Transact-SQL)
 REVERSE (Transact-SQL)
 RIGHT (Transact-SQL)
 RTRIM (Transact-SQL)
 SOUNDEX (Transact-SQL)
 SPACE (Transact-SQL)
 STR (Transact-SQL)

文字列関数 (Transact-SQL) ×

文字列関数 (Transact-SQL)

このトピックに関するフィードバック

以下のスカラー関数は、文字列型の入力値に対して操作を行い、文字列値または数値を返します。

ASCII	LTRIM	SOUNDEX
CHAR	NCHAR	SPACE
CHARINDEX	PATINDEX	STR
CONCAT	QUOTENAME	STUFF
DIFFERENCE	REPLACE	SUBSTRING

6.4 数値操作の関数

➡ 数値操作の関数

数値操作が行える関数には、主に次のものが用意されています。

SQL Server	役割
ROUND	四捨五入
POWER	べき乗
RAND	乱数
CEILING(n)	n に対してそれ以上の最小の整数
FLOOR(n)	n に対してそれ以下の最大の整数
SQRT	平方根

➡ Let's Try

1. まずは、次のように **ROUND** 関数を利用して、「1.66666」という値を小数点以下 **3 桁** になるように四捨五入しましょう。

```
SELECT ROUND(1.66666, 3)
```

The screenshot shows a SQL query window with the text 'SELECT ROUND(1.66666, 3)'. Below the query, the 'Results' tab is active, displaying a single row with the value '1.66700' under the column header '(列名なし)'.

	(列名なし)
1	1.66700

ROUND 関数を使うと、Excel のワークシート関数の ROUND と同じように、四捨五入（算術型丸め）をすることができます。第 2 引数へ四捨五入を行いたい桁を指定します。この例のように、「3」と指定した場合は、小数点以下 3 桁になるように四捨五入を行います。

2. 次に、**RAND** 関数を利用して、乱数を取得してみましょう。

```
SELECT RAND()
```

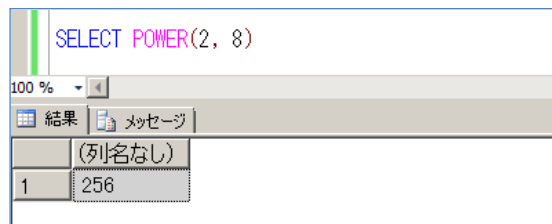
The screenshot shows a SQL query window with the text 'SELECT RAND()'. Below the query, the 'Results' tab is active, displaying a single row with a random decimal value '0.603471144718494' under the column header '(列名なし)'.

	(列名なし)
1	0.603471144718494

RAND 関数では、0 から 1 までの範囲の乱数（float 型）を取得することができます。

3. 次に、**POWER** 関数を利用して、2 の 8 乗を計算してみましょう。

```
SELECT POWER(2, 8)
```



	(列名なし)
1	256

POWER 関数では、指定した値の n 乗を計算することができます。

6.5 NULL 操作の関数 (ISNULL、COALESCE)

➡ NULL 操作の関数 (ISNULL、COALESCE)

SQL Server では、NULL 値を操作するための関数として **ISNULL** 関数が用意されています。ISNULL 関数を利用すると、NULL 値を別の値へ置き換えることができます。構文は次のとおりです。

ISNULL (数値データ , 置換後のデータ)

➡ Let's Try

それでは、これを試してみましょう。

1. 次のように **ISNULL** 関数を利用して、「emp」テーブルの「sal」（給与）列に NULL 値があった場合に、0 へ置き換えるようにします。

```
SELECT ISNULL( sal, 0 ), * FROM emp
```

	(列名なし)	empno	empname	sal	hiredate	deptno
1	500000	1	鈴木 一郎	500000	1998-04-01 00:00:00.000	10
2	300000	2	小田 良夫	300000	1999-04-01 00:00:00.000	10
3	0	3	浅田 ゆかり	NULL	NULL	20
4	700000	4	田村 健一	700000	1985-04-01 00:00:00.000	10
5	500000	5	中野 浩之	500000	1996-04-01 00:00:00.000	10
6	500000	6	内藤 太郎	500000	1997-04-01 00:00:00.000	10
7	250000	7	大和田 聡	250000	2000-04-01 00:00:00.000	20
8	300000	8	長谷川 忍	300000	2006-04-01 00:00:00.000	20
9	600000	9	Geof Cruise	600000	2006-04-01 00:00:00.000	20
10	9999	11	xxx	9999	2006-04-01 08:55:30.000	20
11	9999	12	yyy	9999	2012-07-04 02:27:10.363	20

第 2 引数で “0” を指定しているので、「sal」列が NULL 値だった場合に “0” へ置換することができます。

Note : ISNULL 関数は Oracle の NVL 関数

ISNULL 関数は、Oracle での NVL 関数に相当する機能です。また、SQL Server と Oracle のどちらでも利用できる関数として **COALESCE** もあり、ISNULL と同じように「**COALESCE(単価, 0)**」と記述して利用できます（結果も同様）。そのほかの Oracle との関数比較については、STEP 6.9 でまとめています。

6.6 IIF 関数による条件分岐

➡ IIF 関数による条件分岐

IIF は、SQL Server 2012 からサポートされた関数で、CASE 式と同じように条件分岐が行えます。構文は次のとおりです。

IIF(条件式, 条件を満たした場合の値, 条件を満たしていない場合の値)

➡ Let's Try

それでは、これを試してみましょう。

1. 次のように **IIF** 関数を利用して、「**emp**」テーブルの「**sal**」（給与）列の値が、50 万円以上の場合には「50 万円以上」、そうでない場合は「50 万円未満」と表示するようにします。

```
SELECT IIF( sal >= 500000, '50万円以上', '50万円未満' ), sal FROM emp
```

	(列名なし)	sal
1	50万円以上	500000
2	50万円未満	300000
3	50万円未満	NULL
4	50万円以上	700000
5	50万円以上	500000
6	50万円以上	500000
7	50万円未満	250000
8	50万円未満	300000
9	50万円以上	600000

給与が 50 万 円以上の場合は「50 万円以上」、そうでない場合は「50 万円未満」と表示されることを確認できます。

Note : IIF 関数は Oracle の DECODE 関数

IIF 関数は、Oracle での DECODE、Access での IIF 関数に相当します。

6.7 CHOOSE 関数による指定した値の取得

➡ CHOOSE 関数による指定した値の取得

CHOOSE も SQL Server 2012 からサポートされた関数で、引数で指定した値を取得できる便利な関数です。構文は次のとおりです。

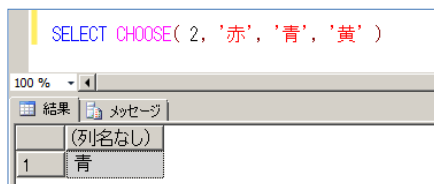
```
CHOOSE ( 取得する値の番号, 値1, 値2, ... )
```

➡ 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 **CHOOSE** 関数を利用してみます。

```
SELECT CHOOSE ( 2, '赤', '青', '黄' )
```



(列名なし)
1 青

CHOOSE 関数では、第 2 引数以降で指定した値リストの中から、第 1 引数で指定した値 (2 の場合は **2 番目の値**) を取得できるので、「青」を取得できています。第 1 引数に 1 を指定した場合は「赤」、3 の場合は「黄」を取得できます。

2. 次に、次のように **CHOOSE** 関数を利用して、「emp」テーブルの「hiredate」(入社日) から曜日を取得することもできます

```
SELECT CHOOSE (DATEPART (weekday, hiredate), '日', '月', '火', '水', '木', '金', '土'),  
DATEPART (weekday, hiredate), hiredate FROM emp
```

(列名なし)	(列名なし)	hiredate
水	4	1998-04-01 00:00:00.000
木	5	1999-04-01 00:00:00.000
NULL	NULL	NULL
月	2	1985-04-01 00:00:00.000
月	2	1996-04-01 00:00:00.000
火	3	1997-04-01 00:00:00.000
土	7	2000-04-01 00:00:00.000
土	7	2006-04-01 00:00:00.000
土	7	2006-04-01 00:00:00.000

CHOOSE の第 1 引数に「**DATEPART (weekday, hiredate)**」を記述して **emp** テーブルの **hiredate** から曜日番号 (1 は日曜、2 は月曜、3 は火曜、…) を取得し、1 だったら「日」(CHOOSE の第 2 引数)、2 だったら「月」(第 3 引数) を返すようにしています。

6.8 ユーザー定義関数

➡ ユーザー定義関数

関数は、ユーザー自身が作成（定義）することもできます。このような関数を**ユーザー定義関数**（UDF : User Defined Function）と呼んでいます。

ユーザー定義関数を作成するには、「**CREATE FUNCTION**」というステートメントを使用します。構文は、次のとおりです。

```
CREATE FUNCTION 関数名 ( [@パラメーター名 データ型], ... )
RETURNS 戻り値のデータ型
[AS]
BEGIN
    ステートメント
    RETURN 戻り値
END
```

パラメーターと AS は省略可能です。BEGIN ～ END で囲まれた領域の中へ処理したい内容を記述し、“RETURN” に続けて戻り値を記述します。

➡ Let's Try

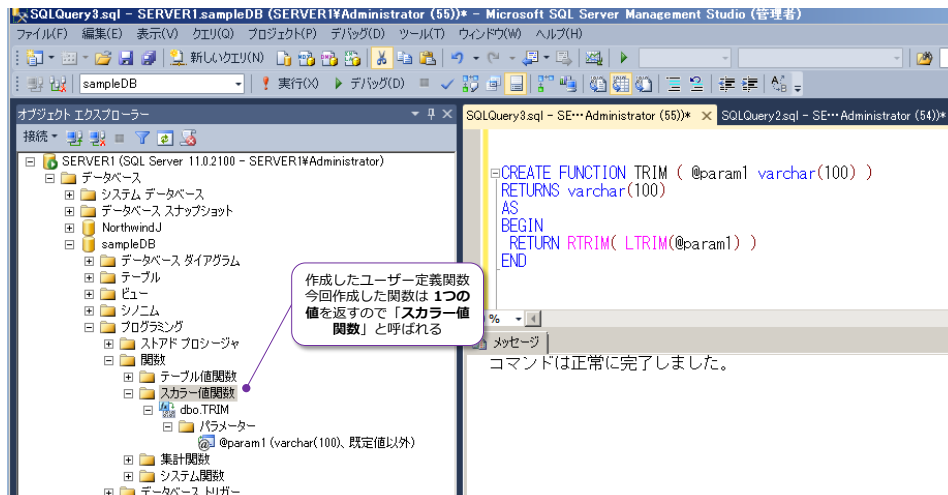
それでは、ユーザー定義関数を作成してみましょう。

1. 次のように左右の半角スペースを除去する「TRIM」関数を作成します。

```
USE sampleDB
go
CREATE FUNCTION TRIM ( @param1 varchar(100) )
RETURNS varchar(100)
AS
BEGIN
    RETURN RTRIM( LTRIM(@param1) )
END
```

関数の名前は“TRIM”とし、“@param1”という名前のパラメーター（引数）を 1 つ作り、データ型を“**varchar(100)**”にしています（100 バイト以上の文字列を扱わせたい場合は、100 の部分を変更してください）。戻り値のデータ型は、パラメーターと同じ“**varchar(100)**”にし、BEGIN ～ END で囲まれた領域へ、左右の半角スペースを除去する処理を記述しています。これは前述したように **RTRIM** 関数と **LTRIM** 関数を入れ子にしています。

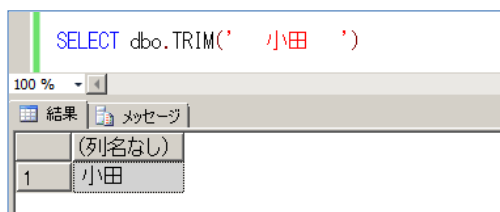
2. 次に、作成したユーザー定義関数をオブジェクト エクスプローラーで確認します。



名前が「**dbo.TRIM**」と、「**dbo.**」が付いていることに注目します。**dbo** は、**DataBase Owner** の略で、データベースの所有者という意味です。SQL Server の管理者アカウントを利用して、SQL Server へ接続している場合には、自動的に **dbo** としてデータベースを操作しています。したがって、**dbo** が作成した **TRIM** 関数ということで「**dbo.TRIM**」となります。

3. 続いて、次のように入力して、作成したユーザー定義関数を使用してみましょう。

```
SELECT dbo.TRIM(' 小田 ')
```



結果は、左右の半角スペースが削除されることを確認できます。**@param1** へ “ 小田 ” を与え、関数の中では「**RTRIM(LTRIM(@param1))**」が実行されているという形です。

4. 続いて、「**emp**」テーブルの「**empname**」列に対して、「**dbo.TRIM**」関数を実行してみましょう。

```
SELECT dbo.TRIM(empname) + 'さん', * FROM emp
```

	(列名なし)	empno	empname	sal	hiredate	deptno
1	鈴木 一郎さん	1	鈴木 一郎	500000	1998-04-01 00:00:00.000	10
2	小田 良夫さん	2	小田 良夫	300000	1999-04-01 00:00:00.000	10
3	浅田 ゆかりさん	3	浅田 ゆかり	NULL	NULL	20
4	田村 健一さん	4	田村 健一	700000	1985-04-01 00:00:00.000	10
5	中野 浩之さん	5	中野 浩之	500000	1996-04-01 00:00:00.000	10
6	内藤 太郎さん	6	内藤 太郎	500000	1997-04-01 00:00:00.000	10
7	大和田 聡さん	7	大和田 聡	250000	2000-04-01 00:00:00.000	20
8	長谷川 忍さん	8	長谷川 忍	300000	2006-04-01 00:00:00.000	20
9	Geof Cruiseさん	9	Geof Cruise	600000	2006-04-01 00:00:00.000	20
10	xxxさん	11	xxx	9999	2006-04-01 08:55:30.000	20
11	yyyさん	12	yyy	9999	2012-07-04 02:27:10.363	20

char(50) で定義された「empname」列の右側の余分な空白が、TRIM 関数によって削除できていることを確認できます。

Note : .NET Framework 言語を使ったユーザー定義関数 (SQLCLR)

ユーザー定義関数 (UDF) は、Visual Basic や C# などの .NET Framework 言語を使っても作成することができます。たとえば、Transact-SQL の ROUND 関数は四捨五入 (算術型丸め) ですが、.NET Framework 言語には、「銀行型丸め」を実装している「Math.Round」メソッドが用意されているので、このメソッドを呼び出すようなユーザー定義関数も簡単に作成することができます。

なお、.NET Framework 言語で作成した UDF は「**SQLCLR 関数**」とも呼ばれます。

➡ Goal !

以上ですべての操作が完了です。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？ 今回は、入門編ということで、Transact-SQL の基本的な操作方法のご紹介のみになりました。動的 SQL (sp_executesql) やストアド プロシージャの作成方法、トランザクション、エラー処理といった応用的な利用方法については、本自習書シリーズの「**開発者のための Transact-SQL 応用**」で説明していますので、ぜひチャレンジしてみてください

6.9 Oracle の関数との比較

➡ Oracle の関数との比較

関数は、データベース製品によって実装が異なります。特に日付や文字列の扱いとデータ型の変換は、製品によって大きく異なるので注意してください。以下は、Oracle の関数と比較した SQL Server の関数です。

日付と時刻、型変換の関数

	SQL Server	Oracle	役割
日付と時刻	GETDATE/CURRENT_TIMESTAMP	CURRENT_TIMESTAMP	現在の日付と時刻
	DATEADD	日付±n ADD_MONTHS	日付の加減算
	DATEPART、YEAR、MONTH	TO_CHAR(日付データ, 書式) EXTRACT(書式 FROM 日付データ)	文字列変換、部分抽出
	EOMONTH	LAST_DAY	月の最終日
	DATEDIFF	日付 - 日付 MONTHS_BETWEEN	日付の差分
変換	CONVERT(char(n), 日付データ, x) FORMAT(日付データ, 書式)	TO_CHAR(日付データ, 書式)	日付を文字列変換
	CONVERT(char(n), 数値データ) FORMAT(数値データ, 書式)	TO_CHAR(数値データ, 書式)	数値を文字列変換
	CONVERT(decimal(p,s), 文字データ)	TO_NUMBER(文字データ, 書式)	文字列を数値変換
	CONVERT(datetime, 文字データ) DATEFROMPARTS	TO_DATE(文字データ, 書式)	文字列を日付変換

文字列関数

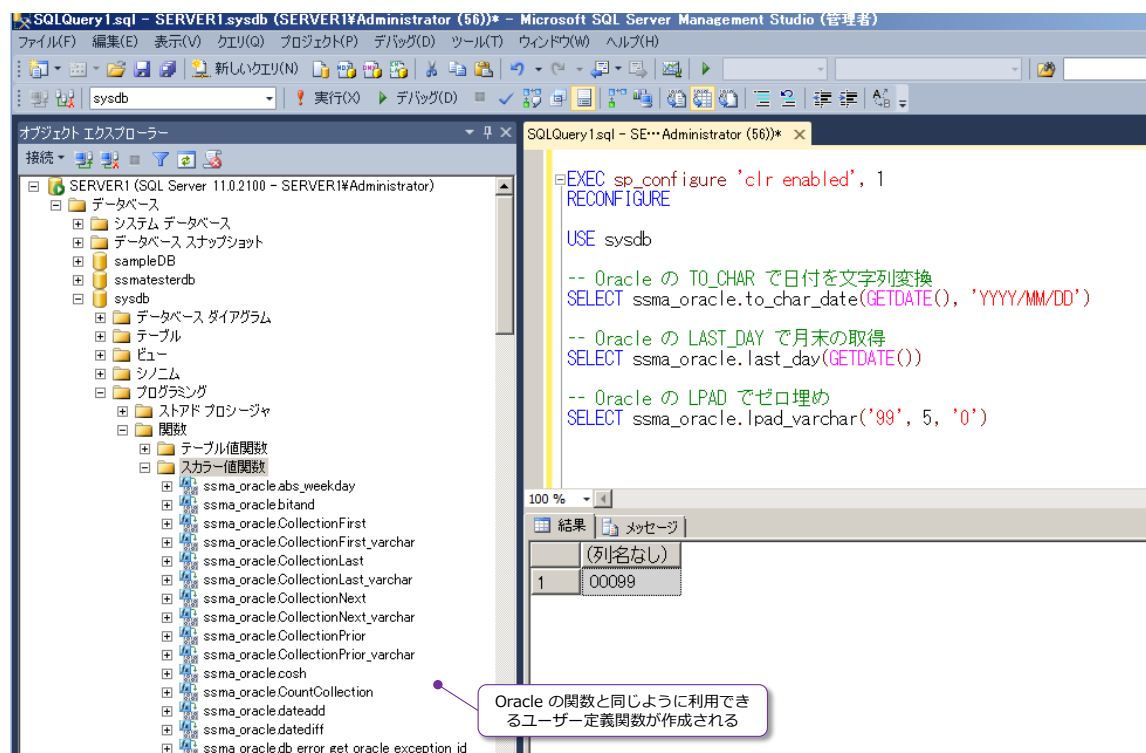
SQL Server	Oracle	役割
CONCAT / +	CONCAT /	文字列連結
UPPER/LOWER	同じ	大文字、小文字変換
RTRIM/LTRIM	同じ	右/左から半角スペースの削除 Oracle ではスペース以外の文字も指定可能
なし	TRIM	左右の指定文字削除
REPLACE/STUFF	REPLACE	文字列の置換
RIGHT/LEFT	なし	右/左から部分抽出
SUBSTRING	SUBSTR	部分抽出
LEN/DATALENGTH	LENGTH/LENGTHB	文字列の長さ、バイト数
CHARINDEX	INSTR	文字列内の検索
なし	INITCAP	文字列の先頭を大文字変換
なし	RPAD/LPAD	右/左に指定文字を埋める
ASCII	同じ	文字の ASCII コード取得
CHAR	CHR	ASCII コードを文字変換

数値関数

SQL Server	Oracle	役割
%	MOD	剰余
ROUND	ROUND	四捨五入
なし	TRUNC	切り捨て SQL Server では ROUND の第3引数に 1 を指定
POWER	POWER	べき乗
RAND	なし	乱数
CEILING(n)	CEIL(n)	n に対してそれ以上の最小の整数
FLOOR(n)	FLOOR(n)	n に対してそれ以下の最大の整数
SQRT	SQRT	平方根
SIN/COS/TAN	同じ	

◆ SSMA for Oracle V5.2 による Oracle 対応関数の提供

SQL Server Migration Assistant for Oracle V5.2 (SSMA for Oracle V5.2) は、Oracle から SQL Server 2012 への移行 (Migration) を支援する無償ツールです。このツールを利用すると、Oracle の関数 (**TO_CHAR** や **LAST_DAY**、**LPAD**、**TRIM**、**MOD** など) と同じように利用できるユーザー定義関数を作成したり、データベースの移行 (転送) が行えたりする非常に便利なツールです。



SQL Server Migration Assistant for Oracle V5.2 のダウンロードはこちらから行えます。

<http://www.microsoft.com/en-us/download/details.aspx?id=28766>

また、古いバージョンの情報になりますが、次の URL からダウンロードできる自習書が参考になると思います。

SQL Server Migration Assistant for Oracle V3.1 自習書

<http://technet.microsoft.com/ja-jp/sqlserver/gg638878.aspx>

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 などでも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。