

はじめに

本書は、PS ネットワークより平成 17 年 6 月に初版が出版された『Pocket Tech Note Firebird RDBMS 傾向と対策 4』の原稿を基にして PDF 化したものです。出版の際には B6 サイズの横長の体裁でしたが、PDF 化するにあたって、A4 縦のサイズへと変更し、必要最低限の編集を加えています。昨年 PDF 化して公開した『Pocket Tech Note Firebird RDBMS 傾向と対策 2』に引き続き、公開をすることとなりました。本書刊行時には末尾に「システムテーブル一覧」を収録していましたが、こちらはテキストデータから版組を行った関係で本 PDF への収録は見送っています。

本書の内容は、Firebird の 1.0.2 がリリースされた当初のものですが、基本的な SQL の構文やデータ型に関する知識は、いまでも大きく変わっていません。管理ツールとしてご紹介した IBOConsole や Marathon はその後開発が中断しており、現在では Firebird Project 公式の管理ツールとして FlameRobin が開発・公開されています。Firebird 日本ユーザー会の加藤大受氏が日本語を行っており、Firebird 日本ユーザー会のページからもダウンロードすることが出来ますので、今後はこちらをお使い下さい。

2007 年 11 月に版元であった PS ネットワークの武田浩一氏が若くして急逝され、PS ネットワークが廃業となったため、多少でも Firebird の普及のお役に立てればとの想いから、ここに執筆者としての責任においてこれを PDF 化してお配りすることにいたしました。Delphi マガジンや nifty serve の FDELPHI でご活躍された武田浩一さんが、Firebird の普及にも尽力されていたことに感謝するとともに、ご冥福をお祈り申し上げます。(2009 年 12 月 25 日ウルトラの街にて 林 務)

まえがき

前著「Firebird RDBMS 傾向と対策 3」に引き続いて、オープンソースの RDBMS、Firebird のユーザー定義関数(UDF)に関する解説をさせていただきます。本書では Firebird に標準で付属している `ib_udf` と `fbudf` 及びフリーソフト・オープンソース等の UDF ライブラリに含まれる UDF の使い方を解説し、引き続き UDF 作成についての解説を行っています。前著では Firebird のプロシージャ・トリガー言語である PSQL の解説と、ストアードプロシージャ・トリガーの作成方法を示しましたが、Firebird の真の実力を引き出すためには、UDF の利用を避けて通ることは出来ません。

他の商用 RDBMS も同様の機能を各種提供していますが、Firebird では C や Delphi 等のプログラミング言語によって作成した DLL 等の共有ライブラリを関数として SQL から利用することを可能としています。DLL に出来るモノであればある種なんでもありという UDF ですが、一歩間違えるとセキュリティホールとなったり、システム障害を引き起こす可能性もあります。十分に注意して利用するようにしたいものです。

最後になりましたが、このシリーズを支えてくださっている読者のみなさんに感謝いたします。中でも、Firebird の発展を第一に考えて下さっている PS ネットワークの武田さんに感謝の意を表したいと思います。また、世界中で Firebird を支えている開発者の皆さん、ユーザー・コミュニティの皆さんに感謝致します。そして、仕事から帰って家事もあまり手伝わずに原稿を書いてばかりいる私を、常に叱咤激励してくれる妻に感謝します。ビバ!! Firebird!!

目次

はじめに	1
まえがき	1
<i>Yet another Open Source Database</i>	1
FIREBIRD RDBMS 傾向と対策 4	1
UDF - FIREBIRD のユーザー定義関数	4
標準 UDF の解説	5
IB_UDF リファレンス	5
<i>a b s</i> (引数)	6
<i>a c o s</i> (引数)	7
<i>a s c i i _ c h a r</i> (引数)	8
<i>a s c i i _ v a l</i> (引数)	9
<i>a s i n</i> (引数)	10
<i>a t a n</i> (引数)	11
<i>a t a n 2</i> (引数1, 引数2)	12
<i>b i n _ a n d</i> (引数1, 引数2)	13
<i>b i n _ o r</i> (引数1, 引数2)	14
<i>b i n _ x o r</i> (引数1, 引数2)	15
<i>c e i l i n g</i> (引数)	16
<i>c o s</i> (引数)	17
<i>c o s h</i> (引数)	18
<i>c o t</i> (引数)	19
<i>d i v</i> (引数1, 引数2)	20
<i>f l o o r</i> (引数)	21
<i>l n</i> (引数)	22
<i>l o g</i> (引数1, 引数2)	23
<i>l o g 1 0</i> (引数)	25
<i>l o w e r</i> (引数)	26
<i>l t r i m</i> (引数)	27
<i>m o d</i> (引数1, 引数2)	28
<i>p i</i>	29
<i>r a n d</i>	30
<i>r t r i m</i> (引数)	31
<i>s i g n</i> (引数)	32
<i>s i n</i> (引数)	33
<i>s i n h</i> (引数)	34
<i>s q r t</i> (引数)	35
<i>s u b s t r</i> (引数1, 引数2, 引数3)	36
<i>s u b s t r l e n</i> (引数1, 引数2, 引数3)	37
<i>s t r l e n</i> (引数)	37
<i>t a n</i> (引数)	38
<i>t a n h</i> (引数)	39
FBUDF リファレンス	40
<i>i d N v l</i> (引数1, 引数2)	41
<i>S N V L</i> (引数1, 引数2) --- 文字列型用	44
<i>I N U L L I F</i> (引数1, 引数2)	45
<i>D N U L L I F</i> (引数1, 引数2)	46
<i>I 6 4 N U L L I F</i> (引数1, 引数2)	47
<i>S N U L L I F</i> (引数1, 引数2, 戻り値)	48
<i>D O W</i> (引数)	49
<i>S D O W</i> (引数)	50
<i>s r i g h t</i> (引数1, 引数2)	51
<i>a d d D a y</i> (引数1, 引数2)	52
<i>a d d W e e k</i> (引数1, 引数2)	53
<i>a d d M o n t h</i> (引数1, 引数2)	54
<i>a d d Y e a r</i> (引数1, 引数2)	55
<i>a d d M l l i S e c o n d</i> (引数1, 引数2)	56
<i>a d d S e c o n d</i> (引数1, 引数2)	57
<i>a d d M i n u t e</i> (引数1, 引数2)	58
<i>a d d H o u r</i> (引数1, 引数2)	59
<i>g e t E x a c t T i m e s t a m p</i>	60
<i>t r u n c a t e</i> (引数)	61
<i>i 6 4 T r u n c a t e</i> (引数)	63
<i>r o u n d</i> (引数)	65
<i>i 6 4 R o u n d</i> (引数)	67
<i>d p o w e r</i> (引数1, 引数2)	69

string2blob (引数)	70
フリーソフト・オープンソース UDF 編	71
XLibUDF	73
StrTrim(引数)	74
StrUCase(引数)	75
StrLCase(引数)	76
StrFirst(引数 1, 引数 2)	77
StrLast(引数 1, 引数 2)	78
ANSILike(引数 1, 引数 2)	79
CreateUID	79
WriteDebug(引数)	80
TimeStamp()	81
UUIDUDF	82
GUID_CREATE	83
GUID_TO_UUID (引数)	84
UUID_CREATE	85
UUID_TO_GUID(引数)	86
EXTERNAL FILE UDF'S	87
NotNull(引数)	89
File_Write(引数 1, 引数 2)	90
File_WriteLn(引数 1, 引数 2)	91
File_delete(引数)	92
File_create(引数)	93
Strformat(引数 1, 引数 2, 引数 3)	94
File_rename(引数 1, 引数 2)	95
File_concat(引数 1, 引数 2)	96
Padleft(引数 1, 引数 2)	97
Padright(引数 1, 引数 2)	98
Squotedstr(引数)	99
dquotedstr(引数)	100
datetimeformat(引数 1, 引数 2)	101
Numberformat(引数 1, 引数 2)	102
External File UDF's 付属ストアプロシージャ	103
UDF 作成編	106
UDF 作成の基礎	106
引数と戻り値	108
引数を渡す方法(数値型)	108
引数を渡す方法(文字型)	110
文字型のデータを戻り値として返す(1)	111
文字型のデータを戻り値として返す(2)	113
日付時刻型の扱い	114
UDF で BLOB を扱う	116
BLOB UDF のサンプル(1)	118
BLOB UDF のサンプル(2)	119
BLOB フィルタ	123
Blob フィルタのスケルトン	123
APPENDIX A FIREBIRD のデータ型	127
数値データ型	127
整数データ型	127
浮動小数点データ型	127
固定小数点データ型	127
数値データ型の演算上の注意点	127
文字データ型	128
日付時刻型	128
BLOB データ型	128
APPENDIX B FIREBIRD データ型の変換可能性一覧表	129
APPENDIX C FIREBIRD システムテーブル一覧	エラー! ブックマークが定義されていません。

UDF - Firebird ユーザー定義関数

Firebird は標準で、AVG()・COUNT()・MAX()・MIN()・SUM()の各集合関数と、CAST()・EXTRACT()・UPPER()・SUBSTRING()の各変換関数、またジェネレータ関数 GEN_ID()を内部関数として提供しています。他の RDBMS と比較してあまりにも種類が少ないため、最初はとまどわれる方も多いかと思います。

Firebird は、C や Delphi 等のプログラミング言語によって作成した DLL 等の共有ライブラリを関数として SQL から利用することを可能とするユーザー定義関数(UDF)という仕組みがあるため、内部関数は本当に必要不可欠な基本機能だけに限定されています。

しかし、Firebird をインストールすると標準 UDF として `ib_udf` と `fbudf` の二つの共有ライブラリがインストールされるので、同時にインストールされる UDF 登録用の SQL スクリプトを使用することで、直ちに `ib_udf` で 34 種類、`fbudf` で 24 種類もの関数を利用することが可能となっています。

`ib_udf` は、InterBase6 から引き継いだ UDF ライブラリですが、`substr()`の修正と、`substrlen()`の追加が Claudio Varierra 氏によって行われています。三角関数や指数関数、対数関数と文字列操作に関わる基本的な関数群を提供しています。

他方、`fbudf` は Firebird プロジェクトによる独自の UDF ライブラリで InterBase には付属していません。

本書では、この 2 つの標準 UDF ライブラリに含まれている関数群の利用方法についての解説を行います。

また、UDF はその名の通りユーザーが独自に作成することも可能であり、その応用範囲は無限とも言えるモノがあります。本書では、さわりだけをご紹介しますとどめですが、続編にて詳しくご紹介する予定です。

各 UDF には、簡単な解説と実行例、登録用スクリプトを示しました。登録用スクリプトはそれぞれ、`ib_udf.sql` と `fbudf.sql` という名前前で UDF ライブラリがインストールされているディレクトリと一緒にインストールされています。

実行例は、Windows 版 Firebird 1.03 を Windows2000 Professional SP4 にインストールして、`isql` で実行した結果を示したものです。その際、登録用スクリプトを実行して、一括してデータベースで利用可能としましたが、`ib_udf.sql` の `strlen` で `CSTRING(32767)` と定義されている個所がエラーとなりました。

これは、`CREATE DATABASE` 文でデータベースを作成する際に、`CHARACTER SET` を `SJIS_0208` としていたため、Firebird の文字型の長さ制限をオーバーしてしまったためです。`SJIS_0208` や `EUCJ_0208` 等のマルチバイトキャラクタセットを利用する場合は、`32767` の半分である `16383` を指定して下さい。`UNICODE_FSS` の場合は、`1/3` である `10922` を指定して下さい。

引数と戻り値にはそれぞれデータ型を示しましたが、`Integer`・`Double Precision` 等の Firebird データ型とは別に、`CSTRING` 型を C 文字列型として示しました。`CSTRING` 型は、Firebird と UDF のデータの受け渡しで利用する、C 文字列型であり、`Char`・`VarChar`・文字列リテラルを代入することが可能です。また、C 文字列型を受ける場合にも同様に、`Char`・`VarChar` へ代入することが可能です。

Firebird プロジェクトのホームページ

<http://firebird.sourceforge.net/>

株アペックスのホームページ

<http://www.apex-jp.com/>

※表記方法に関して

{ } : カッコで囲まれた要素から一つを選択します

[] : カッコで囲まれた要素を使用する事が出来ます

| : 複数の要素の区切として使用しています。

<> : 詳細な情報を別途表記しています。

!!FB!! : Firebird 独自の拡張を解説しています。

標準 UDF の解説

Firebird には、`ib_udf` と `fbudf` という 2 つの UDF ライブラリが標準で付属しています。これらはそれぞれ、`ib_udf.sql` と `fbudf.sql` という登録用スクリプトを `isql` 等で実行することによって各データベースに登録することが出来ます。

ib_udf リファレンス

`ib_udf` ライブラリは、Firebird が InterBase から引き継いだ UDF ライブラリです。InterBase6 のソースコードをベースにしながらバグフィックスと機能の追加が行われています。

`ib_udf` の引数と戻りについては、基本的に Firebird データ型を使用していますが、文字列の受け渡しには `CString` 型が使用されています。`CString` 型は ASCII 文字列を受け入れるデータ型なので、`SJIS_0208` や `UNICODE_FSS` 等のマルチバイト文字列を受け渡すことが出来ません。

各データ型には巻末で示した **APENDIX B Firebird** データ型の変換可能性一覧表に従って、交換性のあるデータ型を受け渡しする事が可能です。C 文字列型には `Char` 型・`VarChar` 型・文字列リテラルを代入することが可能で、受け取る場合は `Char` 型・`VarChar` 型となります。

abs(引数)

引数の絶対値を返します。

引数 : Double Precision

戻り値 : Double Precision

例 :

```
SQL> SELECT ABS(-1.03) FROM RDBSDATABASE;  
          ABS
```

```
=====
```

1.0300000000000000

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION abs  
        DOUBLE PRECISION  
        RETURNS DOUBLE PRECISION BY VALUE  
        ENTRY_POINT 'IB_UDF_abs' MODULE_NAME 'ib_udf';
```

`acos` (引数)

引数の ArcCosine を返します。ArcCosine は Cosine が引数になる角度を示します。つまり Cosine の逆変換です。
引数は、ラジアンで指定し、その範囲は-1 から 1 となります。それ以外の値を指定した場合、エラーが返されます。

引数 : Double Precision --- ラジアン (-1~1)
戻り値 : Double Precision --- ラジアン (-1~1)

例 :

```
SQL> SELECT ACOS(0.707107) FROM RDBSDATABASE;  
          ACOS
```

```
=====
```

```
          0.7853978539484482
```

※0.78539... は 45 度のラジアン、0.707107 はそのコサインです。

```
SQL> SELECT ACOS(1.1) FROM RDBSDATABASE;  
          ACOS
```

```
=====
```

```
          -1.#IND0000000000
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION acos  
          DOUBLE PRECISION  
          RETURNS DOUBLE PRECISION BY VALUE  
          ENTRY_POINT 'IB_UDF_acos' MODULE_NAME 'ib_udf';
```

`ascii_char`(引数)

引数で指定された数値がアスキーコードである文字を 1 文字返します。単純に `Char` にキャストしてるだけなので、ASCII コードの範囲外の値を指定しても文字が返ってくるので注意して下さい。

引数 : Integer
戻り値 : CString(1)

例 :

```
SQL> SELECT ASCII_CHAR(65) FROM RDBSDATABASE;  
ASCII_CHAR  
=====  
A
```

```
SQL> SELECT ASCII_CHAR(321) FROM RDBSDATABASE;  
ASCII_CHAR  
=====  
A
```

```
SQL> SELECT ASCII_CHAR(-200) FROM RDBSDATABASE;  
  
ASCII_CHAR  
=====  
8
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION ascii_char  
    INTEGER  
    RETURNS CSTRING(1) FREE_IT  
    ENTRY_POINT 'IB_UDF_ascii_char' MODULE_NAME 'ib_udf';
```

ソースコード :

```
char *EXPORT IB_UDF_ascii_char( int *a)  
{  
    char *b;  
    b = (char *) ib_util_malloc(2);  
    *b = (char) (*a);  
    /* let us not forget to NULL terminate */  
    b[1] = '\0';  
    return (b);  
}
```

`ascii_val(引数)`

引数で指定した文字のASCII文字コードを返します。

例：

```
SQL> SELECT ASCII_VAL('A') FROM RDBSDATABASE;  
ASCII_VAL  
=====  
65
```

引 数：Char(1)
戻り値：Integer;

```
SQL> SELECT ASCII_VAL(1) FROM RDBSDATABASE;  
ASCII_VAL  
=====  
49
```

登録用スクリプト：

```
DECLARE EXTERNAL FUNCTION ascii_val  
CHAR(1)  
RETURNS INTEGER BY VALUE  
ENTRY_POINT 'IB_UDF_ascii_val' MODULE_NAME 'ib_udf';
```

asin(引数)

引数の ArcSine を返します。ArcSine は Sine が引数になる角度を示します。つまり Sine の逆変換です。
引数は、ラジアンで指定し、その範囲は 1-から 1 となります。それ以外の値を指定した場合、エラーが返されます。

引数 : Double Precision --- ラジアン(-1~1)
戻り値 : Double Precision --- ラジアン(-1~1)

例 :

```
SQL> SELECT ASIN(0.707107) FROM RDBSDATABASE;  
          ASIN  
=====  
          0.7853984728464484
```

※0.78539... は 45 度のラジアン、0.707107 はそのサインです。

```
SQL> SELECT ASIN(1.1) FROM RDBSDATABASE;  
          ASIN  
=====  
          -1.#IND0000000000
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION asin  
        DOUBLE PRECISION  
        RETURNS DOUBLE PRECISION BY VALUE  
        ENTRY_POINT 'IB_UDF_asin' MODULE_NAME 'ib_udf';
```

a t a n (引数)

引数の ArcTangent を返します。ArcTangent は Tangent が引数になる角度を示します。つまり Tangent の逆変換です。戻り値はラジアンで示されます。

引 数 : Double Precision

戻り値 : Double Precision --- ラジアン (-1~1)

例 :

```
SQL> SELECT ATAN(1) FROM RDBSDATABASE;  
          ATAN
```

```
=====
```

0.7853981633974483

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION atan  
        DOUBLE PRECISION  
        RETURNS DOUBLE PRECISION BY VALUE  
        ENTRY_POINT 'IB_UDF_atan' MODULE_NAME 'ib_udf';
```

`atan2` (引数1, 引数2)

`atan2` は、ArcTangent(引数1 / 引数2)を示します。通常、x-y座標系でxを引数1として、yを引数2として与えることによって、ArcTangentを求めることが出来ます。つまり、x-y座標系からラジアンへの変換を行います。

引数1,2 : Double Precision

戻り値 : Double Precision --- ラジアン(-1~1)

例 :

```
SQL> SELECT ATAN2(1, 1) FROM RDBSDATABASE;  
          ATAN2
```

```
=====  
          0.7853981633974483
```

※x-y座標系で(1, 1)は、45度を示しています。0.78539...は45度のラジアンとなります。

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION atan2  
          DOUBLE PRECISION, DOUBLE PRECISION  
          RETURNS DOUBLE PRECISION BY VALUE  
          ENTRY_POINT 'IB_UDF_atan2' MODULE_NAME 'ib_udf';
```

bin_and(引数 1, 引数 2)

bin_and は、引数 1 と引数 2 のバイナリレベルでの AND 演算の結果を返します。

引数 1, 2 : Integer

戻り値 : Integer

例 :

```
SQL> SELECT BIN_AND(1, 1) FROM RDBSDATABASE;  
      BIN_AND
```

```
=====  
              1
```

```
SQL> SELECT BIN_AND(1, 0) FROM RDBSDATABASE;  
      BIN_AND
```

```
=====  
              0
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION bin_and  
      INTEGER, INTEGER  
      RETURNS INTEGER BY VALUE  
      ENTRY_POINT 'IB_UDF_bin_and' MODULE_NAME 'ib_udf';
```

bin_or (引数 1, 引数 2)

bin_or は、引数 1 と引数 2 のバイナリレベルの **OR** 演算の結果を返します。

引数 1, 2 : Integer

戻り値 : Integer

例 :

```
SQL> SELECT BIN_OR(1, 0) FROM RDBSDATABASE;  
      BIN_OR
```

```
=====  
              1
```

```
SQL> SELECT BIN_OR(0, 0) FROM RDBSDATABASE;  
      BIN_OR
```

```
=====  
              0
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION bin_or  
      INTEGER, INTEGER  
      RETURNS INTEGER BY VALUE  
      ENTRY_POINT 'IB_UDF_bin_or' MODULE_NAME 'ib_udf' ;
```

bin_xor (引数 1, 引数 2)

bin_xor は、引数 1 と引数 2 のバイナリレベルの XOR 演算の結果を返します。

引数 1, 2 : Integer

戻り値 : Integer

例 :

```
SQL> SELECT BIN_XOR(1, 1) FROM RDBSDATABASE;  
      BIN_XOR
```

```
=====  
              0
```

```
SQL> SELECT BIN_XOR(1, 0) FROM RDBSDATABASE;  
      BIN_XOR
```

```
=====  
              1
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION bin_xor  
      INTEGER, INTEGER  
      RETURNS INTEGER BY VALUE  
      ENTRY_POINT 'IB_UDF_bin_xor' MODULE_NAME 'ib_udf';
```

ceiling (引数)

ceiling は、引数以上の整数値で最小の数を返します。つまり、切り上げです。ただし、戻り値は **Double Precision** 型となります。

引数 : **Double Precision**

戻り値 : **Double Precision**

例 :

```
SQL> SELECT CEILING(1.56) FROM RDBSDATABASE;  
          CEILING
```

```
=====
```

```
          2.0000000000000000
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION ceiling  
          DOUBLE PRECISION  
          RETURNS DOUBLE PRECISION BY VALUE  
          ENTRY_POINT 'IB_UDF_ceiling' MODULE_NAME 'ib_udf';
```

`cos` (引数)

引数のコサインを返します。

引数はラジアンで指定し、その範囲が-263 から 263 を超えると精度が損なわれるというコメントがされています。

引数 : Double Precision

戻り値 : Double Precision

例 :

```
SQL> SELECT COS(102846.3187005690) FROM RDBSDATABASE;  
COS
```

```
=====
```

-1.0000000000000000

※102846.31...は、32737* π を示しています。

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION cos  
DOUBLE PRECISION  
RETURNS DOUBLE PRECISION BY VALUE  
ENTRY_POINT 'IB_UDF_cos' MODULE_NAME 'ib_udf';
```

c o s h (引数)

cosh は、引数のハイパーボリックコサインを返します。

引数は実数で指定し、その範囲が-263 から 263 を超えると精度が損なわれるというコメントがされています。

引 数 : Double Precision

戻り値 : Double Precision

例 :

```
SQL> SELECT COSH(4) FROM RDBSDATABASE;  
      COSH
```

```
=====
```

```
27.30823283601649
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION cosh  
      DOUBLE PRECISION  
      RETURNS DOUBLE PRECISION BY VALUE  
      ENTRY_POINT 'IB_UDF_cosh' MODULE_NAME 'ib_udf';
```

cot (引数)

引数のコタンジェントを返します。つまり、引数のタンジェントで1を割った値となります。

引数: Double Precision

戻り値: Double Precision

例:

```
SQL> SELECT COT(1) FROM RDBSDATABASE;  
          COT
```

```
=====
```

0.6420926159343308

```
SQL> SELECT COT(0) FROM RDBSDATABASE;  
          COT
```

```
=====
```

1.#INF00000000000

登録用スクリプト:

```
DECLARE EXTERNAL FUNCTION cot  
          DOUBLE PRECISION  
          RETURNS DOUBLE PRECISION BY VALUE  
          ENTRY_POINT 'IB_UDF_cot' MODULE_NAME 'ib_udf';
```

div (引数 1, 引数 2)

引数 1 を引数 2 で割った商を返します。

引数 1, 2 : Integer

戻り値 : Double Precision

例 :

```
SQL> SELECT DIV(10, 2) FROM RDBSDATABASE;  
      DIV
```

```
=====
```

5.0000000000000000

```
SQL> SELECT DIV(11, 2) FROM RDBSDATABASE;  
      DIV
```

```
=====
```

5.0000000000000000

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION div  
      INTEGER, INTEGER  
      RETURNS DOUBLE PRECISION BY VALUE  
      ENTRY_POINT 'IB_UDF_div' MODULE_NAME 'ib_udf';
```

floor (引数)

引数以下の整数で最大の値を返します。つまり切り捨てです。

引数 : Double Precision

戻り値 : Double Precision

例 :

```
SQL> SELECT FLOOR(3.14159) FROM RDBSDATABASE;
          FLOOR
=====
          3.000000000000000
```

```
SQL> SELECT FLOOR(3) FROM RDBSDATABASE;
          FLOOR
=====
          3.000000000000000
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION floor
          DOUBLE PRECISION
          RETURNS DOUBLE PRECISION BY VALUE
          ENTRY_POINT 'IB_UDF_floor' MODULE_NAME 'ib_udf';
```

ln (引数)

引数の自然対数を返します。自然対数とは、**e (2.71828182845904...)** を底とする対数です。

引数 : **Double Precision**

戻り値 : **Double Precision**

例 :

```
SQL> SELECT LN(2.71828182845904) FROM RDBSDATABASE;
```

```
LN
```

```
=====
```

```
0.9999999999999981
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION ln
```

```
DOUBLE PRECISION
```

```
RETURNS DOUBLE PRECISION BY VALUE
```

```
ENTRY_POINT 'IB_UDF_ln' MODULE_NAME 'ib_udf';
```

log (引数 1, 引数 2)

引数 2 を底とする引数 1 の対数を返します。

引数 1, 2 : Double Precision

戻り値 : Double Precision

例 :

```
SQL> SELECT LOG(8, 2) FROM RDBSDATABASE;
```

```
LOG
```

```
=====
```

```
3.0000000000000000
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION log
    DOUBLE PRECISION, DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT 'IB_UDF_log' MODULE_NAME 'ib_udf';
```

※登録用スクリプトのコメントでは、「**log (x, y) returns the logarithm base x of y.**」となっているため、**x**を底とした**y**の対数という意味なのですが、実際には**y**を底とした**x**の対数が返されます。注意して下さい。

この問題はバグとして認識されており、**Firebird v1.5**では修正されています。また、**v1**系列へのバックポートも行われており、メンテナンス・リリースには含まれる模様です。ただ、**Excel**等と、現在の結果は等しいため、コメントを変更するべきだったのではないかと思います。

論旨としては、英語圏では**base x of y**という記述が通常なのでこれに合わせる、また、**log10()**との相似からいっても底が前に来る方がいいということでした。

[Firebird v1.0.3 --- ib_udf.c v1.2のコード]

```
double EXPORT IB_UDF_log(  
    double* a,  
    double* b)  
{  
    return (log(*a)/log(*b));  
}
```

[Firebird v1.5 --- ib_udf.c v1.7のコード]

```
double EXPORT IB_UDF_log( double *a, double *b)  
{  
    return (log(*b) / log(*a));  
}
```

log10 (引数)

10を底とする対数を返します。

引数: Double Precision

戻り値: Double Precision

例:

```
SQL> SELECT LOG10(100) FROM RDBSDATABASE;  
          LOG10
```

```
=====
```

2.0000000000000000

登録用スクリプト:

```
DECLARE EXTERNAL FUNCTION log10  
          DOUBLE PRECISION  
          RETURNS DOUBLE PRECISION BY VALUE  
          ENTRY_POINT 'IB_UDF_log10' MODULE_NAME 'ib_udf';
```

l o w e r (引数)

指定された ASCII 文字列を小文字にして返します。

登録用スクリプトでは、引数と戻り値が **CSTRING(80)** となっていますが、**80** 文字までに制限されているわけではありません。実際は、**32,767** 文字の **Firebird** の文字列の制限まで利用することが可能です。
その場合、登録用スクリプトの文字列長指定を必要な数値に変更して登録して下さい。

引 数 : CString(80)

戻り値 : CString(80)

例 :

```
SQL> SELECT LOWER('ABC') FROM RDBSDATABASE;
```

```
LOWER
```

```
=====abc
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION lower  
    CSTRING(80)  
    RETURNS CSTRING(80) FREE_IT  
    ENTRY_POINT 'IB_UDF_lower' MODULE_NAME 'ib_udf';
```

ltrim (引数)

指定された ASCII 文字列の左側に存在する空白を取り除きます。

登録用スクリプトでは、引数と戻り値が **CSTRING(80)** となっていますが、**80** 文字までに制限されているわけではありません。実際は、**32,767** 文字の **Firebird** の文字列の制限まで利用することが可能です。

その場合、登録用スクリプトの文字列長指定を必要な数値に変更して登録して下さい。

引 数 : **CString(80)**

戻り値 : **CString(80)**

例 :

```
SQL> SELECT LTRIM(' ABC ') FROM RDBSDATABASE;
```

```
LTRIM
```

```
=====
```

ABC

※右側の空白二文字は残っています。

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION ltrim
```

```
    CSTRING(80)
```

```
    RETURNS CSTRING(80) FREE_IT
```

```
    ENTRY_POINT 'IB_UDF_ltrim' MODULE_NAME 'ib_udf';
```

mod (引数 1, 引数 2)

引数 1 を引数 2 で割った余りを返します。

引数 1, 2 : Integer

戻り値 : Double Precision

例 :

```
SQL> SELECT MOD(10, 4) FROM RDBSDATABASE;
```

```
MOD
```

```
=====
```

```
2.0000000000000000
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION mod
```

```
INTEGER, INTEGER
```

```
RETURNS DOUBLE PRECISION BY VALUE
```

```
ENTRY_POINT 'IB_UDF_mod' MODULE_NAME 'ib_udf';
```

pi

円周率を返します。引数は取りません。

引数：無し
戻り値：Double Precision

例：

```
SQL> SELECT PI () FROM RDBSDATABASE;  
PI
```

```
=====
```

3.141592653589793

※引数を取らない場合も()は省略できません。

登録用スクリプト：

```
DECLARE EXTERNAL FUNCTION pi  
    RETURNS DOUBLE PRECISION BY VALUE  
    ENTRY_POINT 'IB_UDF_pi' MODULE_NAME 'ib_udf';
```

rand

0 から 1 の間の乱数を返します。引数は取りません。

また、乱数の初期化に現在の時刻を使用するため、連続した呼出では同じ値が返ってくるので注意して下さい。

引 数：無し

戻り値：Double Precision

例：

```
SQL> SELECT RAND() FROM RDBSDATABASE;  
      RAND
```

```
=====
```

```
0.4143498031556139
```

登録用スクリプト：

```
DECLARE EXTERNAL FUNCTION rand  
  RETURNS DOUBLE PRECISION BY VALUE  
  ENTRY_POINT 'IB_UDF_rand' MODULE_NAME 'ib_udf';
```

`rtrim` (引数)

指定された ASCII 文字列の右側に存在する空白を取り除きます。

登録用スクリプトでは、引数と戻り値が **CSTRING(80)** となっていますが、**80** 文字までに制限されているわけではありません。実際は、**32,767** 文字の **Firebird** の文字列の制限まで利用することが可能です。

その場合、登録用スクリプトの文字列長指定を必要な数値に変更して登録して下さい。

引数 : CSTRING(80)

戻り値 : CSTRING(80)

例 :

```
SQL> SELECT RTRIM(' ABC ') FROM RDBSDATABASE;
```

```
RTRIM
```

```
=====
```

```
ABC
```

※左側の空白二文字は残っています。

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION rtrim
```

```
    CSTRING(80)
```

```
    RETURNS CSTRING(80) FREE_IT
```

```
    ENTRY_POINT 'IB_UDF_rtrim' MODULE_NAME 'ib_udf';
```

sign (引数)

引数が正の場合に 1 を、ゼロの場合に 0 を、負の数の場合に -1 を返します。

引数 : Double Precision

戻り値 : Integer

例 :

```
SQL> SELECT SIGN(-10) FROM RDBSDATABASE;
```

```
      SIGN
```

```
=====
```

```
      -1
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION sign
      DOUBLE PRECISION
      RETURNS INTEGER BY VALUE
      ENTRY_POINT 'IB_UDF_sign' MODULE_NAME 'ib_udf';
```

sin (引数)

引数のサインを返します。

引数はラジアンで指定し、その範囲が-263 から 263 を超えると精度が損なわれるというコメントがされています。

引数 : Double Precision

戻り値 : Double Precision

例 :

```
SQL> SELECT SIN(0.5*PI()) FROM RDBSDATABASE;  
          SIN
```

```
=====  
          1.0000000000000000
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION sin  
        DOUBLE PRECISION  
        RETURNS DOUBLE PRECISION BY VALUE  
        ENTRY_POINT 'IB_UDF_sin' MODULE_NAME 'ib_udf';
```

sinh (引数)

cosh は、引数のハイパーボリックサインを返します。

引数は実数で指定し、その範囲が-263 から 263 を超えると精度が損なわれるというコメントがされています。

引数 : Double Precision

戻り値 : Double Precision

例 :

```
SQL> SELECT SINH(4) FROM RDBSDATABASE;  
        SINH
```

```
=====
```

```
27.28991719712775
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION sinh  
        DOUBLE PRECISION  
        RETURNS DOUBLE PRECISION BY VALUE  
        ENTRY_POINT 'IB_UDF_sinh' MODULE_NAME 'ib_udf';
```

`sqrt` (引数)

引数の平方根を返します。

引数: Double Precision

戻り値: Double Precision

例:

```
SQL> SELECT SQRT(4) FROM RDBSDATABASE;  
          SQRT
```

```
=====
```

2.0000000000000000

登録用スクリプト:

```
DECLARE EXTERNAL FUNCTION sqrt  
        DOUBLE PRECISION  
        RETURNS DOUBLE PRECISION BY VALUE  
        ENTRY_POINT 'IB_UDF_sqrt' MODULE_NAME 'ib_udf';
```

`substr` (引数 1, 引数 2, 引数 3)

引数 1 で指定された文字列の部分文字列を返します。戻り値は、引数 2 で指定された文字から、引数 3 で指定された文字までとなります。引数 3 は部分文字列の長さではありません。

登録用スクリプトでは、引数と戻り値が `CSTRING(80)` となっていますが、80 文字までに制限されているわけではありません。実際は、32,767 文字の Firebird の文字列の制限まで利用することが可能です。

その場合、登録用スクリプトの文字列長指定を必要な数値に変更して登録して下さい。

引数 1 : `CString(80)`

引数 2, 3 : `Smallint`

戻り値 : 文字列型

例 :

```
SQL> SELECT SUBSTR('ABCDE', 2, 3) FROM RDBSDATABASE;  
SUBSTR
```

```
=====
```

```
BC
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION substr  
    CSTRING(80), SMALLINT, SMALLINT  
    RETURNS CSTRING(80) FREE_IT  
    ENTRY_POINT 'IB_UDF_substr' MODULE_NAME 'ib_udf';
```

※InterBase6 付属の `ib_udf` では、引数 3 が引数 1 で指定された文字列の長さより大きい場合には、`NULL` が返されていました。

Firebird の `ib_udf` では、引数 3 が引数 1 の文字列のながさより大きい場合も、引数 2 文字目から最後までが返されるようになっています。

この修正は Claudio Valderrama 氏によって行われました。

[InterBase6 の `ib_udf` のコード]

```
if (length = strlen(s))  
{  
    if (*m > *n || *m < 1 || *n < 1 ||  
        *m > length || *n > length)  
        return NULL;  
    ... 略 ...
```

[Firebird の `ib_udf` のコード]

```
length = strlen(s);  
if (!length || *m > *n || *m < 1 || *n < 1 ||  
    *m > length)  
{  
    buf = (char*) ib_util_malloc(1);  
    buf[0] = '\0'; length = strlen(s);  
    ... 略 ...
```

substrlen (引数 1, 引数 2, 引数 3)

引数 1 で指定された文字列の部分文字列を返します。戻り値は、引数 2 で指定された文字から、引数 3 で指定された長さとなります。

登録用スクリプトでは、引数と戻り値が **CSTRING(80)** となっていますが、**80** 文字までに制限されているわけではありません。実際は、**32,767** 文字の **Firebird** の文字列の制限まで利用することが可能です。

その場合、登録用スクリプトの文字列長指定を必要な数値に変更して登録して下さい。

引数 1 : CString(80)

引数 2, 3 : Smallint

戻り値 : 文字列型

例 :

```
SQL> SELECT SUBSTRLEN('ABCDE', 2, 3) FROM RDBSDATABASE;  
SUBSTRLEN
```

```
=====
```

```
BCD
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION substrlen  
    CSTRING(80), SMALLINT, SMALLINT  
    RETURNS CSTRING(80) FREE_IT  
    ENTRY_POINT 'IB_UDF_substrlen' MODULE_NAME 'ib_udf';
```

strlen (引数)

指定された文字列の長さを返します。

引 数 : CString(80)

戻り値 : Integer

例 :

```
SQL> SELECT STRLEN('ABCDE') FROM RDBSDATABASE;  
STRLEN
```

```
=====
```

```
5
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION strlen  
    CSTRING(32767)  
    RETURNS INTEGER BY VALUE  
    ENTRY_POINT 'IB_UDF_strlen' MODULE_NAME 'ib_udf';
```

`tan` (引数)

引数のタンジェントを返します。引数はラジアンで指定し、その範囲が-263 から 263 を超えると精度が損なわれるというコメントがされています。

引数 : Double Precision

戻り値 : Double Precision

例 :

```
SQL> SELECT TAN(PI()/4) FROM RDBSDATABASE;  
          TAN
```

```
=====
```

0.9999999999999999

```
SQL> SELECT TAN(4*PI()+PI()/4) FROM RDBSDATABASE;  
          TAN
```

```
=====
```

1.0000000000000001

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION tan  
        DOUBLE PRECISION  
        RETURNS DOUBLE PRECISION BY VALUE  
        ENTRY_POINT 'IB_UDF_tan' MODULE_NAME 'ib_udf';
```

t a n h (引数)

引数のハイパーボリックタンジェントを返します。

ハイパーボリックタンジェントは $\tanh(n) = \sinh(n) / \cosh(n)$ で計算される値です。

引 数 : Double Precision

戻り値 : Double Precision

例 :

```
SQL> SELECT TANH(0.5) FROM RDBSDATABASE;  
TANH
```

```
=====  
0.4621171572600097
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION tanh  
DOUBLE PRECISION  
RETURNS DOUBLE PRECISION BY VALUE  
ENTRY_POINT 'IB_UDF_tanh' MODULE_NAME 'ib_udf';
```

fbudf リファレンス

fbudf は、Firebird プロジェクト独自の UDF ライブラリです。fbudf では、Firebird で新たに導入された「by descriptor」構文を使用する関数が導入されています。「by descriptor」構文を使用した場合、引数と戻り値は単純な参照渡しや値渡しではなく、ParamDsc 構造体への参照として渡されています。

ParamDsc 構造体は、ibase.h で定義されているものですが、元をたどると dsc.h で定義され、システムテーブル RDBSFORMATS の RDBSDSCRIPTOR に格納される dsc 構造体と同じものです。

dsc 構造体は、データ型・データ長・スケール・サブタイプ・NULL に関するフラグ等から構成されています。データベースの各列の定義は、この dsc 構造体によって表現され、RDBSFORMATS テーブルに格納されているわけです。

fbudf においては、NULL を扱う関数群を提供する関係から、この dsc 構造体を ibase.h で再定義した上で ParamDsc 構造体として利用するという形式を取っています。

ParamDsc 構造体を利用して、UDF ライブラリへパラメータを渡すためには、Firebird で新たに導入された DESCRIPTOR キーワードを利用します。具体的には、各関数の登録用スクリプトを参照して下さい。

また、この ParamDsc 構造体を利用することで、同じ関数を交換性のある他のデータ型で複数登録することが可能となっています。例えば、冒頭で取り上げる idnvl () 関数は、invl・i64nvl・dnvl の 3 通りの登録が可能となっています。それぞれ、Integer・Int64・Double Precision 型を引数・戻り値として取るものです。

idNvl (引数 1, 引数 2)

引数 1 が NULL である場合、引数 2 を返します。そうでない場合は、引数 1 をそのまま返します。

INVL (引数 1, 引数 2) --- 整数用

引数 1, 2 : Integer

戻り値 : Integer

例 :

```
SQL> SELECT INVL(CAST(NULL AS INTEGER), 12) FROM RDBSDATABASE;
```

```
      INVL  
=====
```

```
      12
```

```
SQL> SELECT INVL(0, 12) FROM RDBSDATABASE;
```

```
      INVL  
=====
```

```
      0
```

登録用スクリプト 1 :

```
declare external function invl  
int by descriptor, int by descriptor  
returns int by descriptor  
entry_point 'idNvl' module_name 'fbudf';
```

I64NVL (引数 1, 引数 2) --- Int64 用

引数 1, 2 : NUMERIC(18, 0)
戻り値 : NUMERIC(18, 0)

例 :

```
SQL> SELECT I64NVL(CAST(12 AS NUMERIC(18, 0)), 1) FROM RDBSDATABASE;  
          I64NVL  
=====
```

12

登録用スクリプト 2 :

```
declare external function i64nvl  
numeric(18,0) by descriptor, numeric(18,0) by descriptor  
returns numeric(18,0) by descriptor  
entry_point 'idNvl' module_name 'fbudf';
```

DNVL(引数 1, 引数 2) --- Double Precision 用

引数 1, 2 : Double Precision

戻り値 : Double Precision

例 :

```
SQL> SELECT DNVL(CAST(12.565 AS DOUBLE PRECISION), 1) FROM RDBSDATABASE;
```

```
      DNVL
```

```
=====
      12.56500000000000
```

```
declare external function dnv1
double precision by descriptor, double precision by descriptor
returns double precision by descriptor
entry_point 'idNvl' module_name 'fbudf';
```

SNVL (引数 1, 引数 2) --- 文字列型用

引数 1 が NULL である場合、引数 2 を返します。そうでない場合は、引数 1 をそのまま返します。

引数 1, 2 : VarChar(100)

戻り値 : VarChar(100)

例 :

登録用スクリプト :

```
declare external function snvl  
varchar(100) by descriptor, varchar(100) by descriptor,  
varchar(100) by descriptor returns parameter 3  
entry_point 'sNvl' module_name 'fbudf';
```

INULLIF(引数 1, 引数 2)

引数 1 が NULL であれば引数 2 を返します。そうでなければ、引数 1 を返します。

引数 1, 2 : Integer

戻り値 : Integer

例 :

登録用スクリプト :

```
declare external function inullif
int by descriptor, int by descriptor
returns int by descriptor
entry_point 'iNullIf' module_name 'fbudf';
```

DNULLIF (引数 1, 引数 2)

引数 1 が NULL であれば引数 2 を返します。そうでなければ、引数 1 を返します。

引数 1, 2 : Double Precision

戻り値 : Double Precision

例 :

登録用スクリプト :

```
declare external function dnullif
double precision by descriptor, double precision by descriptor
returns double precision by descriptor
entry_point 'dNullIf' module_name 'fbudf';
```

I64NULLIF (引数 1, 引数 2)

引数 1 が NULL であれば引数 2 を返します。そうでなければ、引数 1 を返します。

引数 1, 2 : Numeric(18, 0)

戻り値 : Numeric(18, 0)

例 :

登録用スクリプト :

```
declare external function i64nullif
numeric(18,4) by descriptor, numeric(18,4) by descriptor
returns numeric(18,4) by descriptor
entry_point 'iNullIf' module_name 'fbudf';
```

SNULLIF (引数 1, 引数 2, 戻り値)

引数 1 が NULL であれば引数 2 を返します。そうでなければ、引数 1 を返します。
戻り値は 3 番目の引数へ代入して返されるので注意して下さい。

引数 1, 2 : VarChar(100)

戻り値 : VarChar(100)

例 :

登録用スクリプト :

```
declare external function snullif  
varchar(100) by descriptor, varchar(100) by descriptor,  
varchar(100) by descriptor returns parameter 3  
entry_point 'sNullif' module_name 'fbudf';
```

DOW (引数)

指定された日付の曜日を OS の設定に従って、長い形式で返します。戻り値は各言語に対応してローカライズされます。

引 数 : 日付

戻り値 : VarChar(15)

例 :

```
SQL> SELECT DOW('2004/1/1') FROM RDBSDATABASE;
```

DOW

```
=====
```

木曜日

登録用スクリプト :

```
declare external function dow  
timestamp,  
varchar(15) returns parameter 2  
entry_point 'DOW module_name 'fbudf';
```

SDOW (引数)

指定された日付の曜日を OS の設定に従って、短い形式で返します。戻り値は各言語に対応してローカライズされます。

引 数 : 日付
戻り値 : VarChar(5)

例 :

```
SQL> SELECT SDOW('2004/1/1') FROM RDBSDATABASE;  
SDOW
```

=====

木

登録用スクリプト :

```
declare external function sdown  
timestamp,  
varchar(5) returns parameter 2  
entry_point 'SDOW' module_name 'fbudf';
```

sright (引数 1, 引数 2)

引数 1 で指定された文字列のうち、右側から引数 2 で指定された長さに相当する部分を返します。引数 2 が文字列の長さを超える場合は、文字列全体を返します。

引数 1 : VarChar(100)
引数 2 : SMALLINT
戻り値 : VarChar(100)

例 :

```
SQL> SELECT SRIGHT('ABCDEF', 2) FROM RDBSDATABASE;  
SRIGHT
```

```
=====
```

```
SQL> SELECT SRIGHT('ABCDEF', 8) FROM RDBSDATABASE;  
SRIGHT
```

```
=====
```

登録用スクリプト :

```
declare external function sright  
varchar(100) by descriptor, smallint,  
varchar(100) by descriptor returns parameter 3  
entry_point 'right' module_name 'fbudf';
```

addDay (引数 1, 引数 2)

引数 1 で指定された日付時刻型に、引数 2 で指定された日数を加算して返します。

引数 1 : **TIMESTAMP** 型
引数 2 : **INTEGER** 型
戻り値 : **TIMESTAMP** 型

例 :

```
SQL> SELECT ADDDAY('2004/1/1', 9) FROM RDBSDATABASE;  
          ADDDAY
```

```
=====
```

```
2004-01-10 00:00:00.0000
```

```
SQL> SELECT ADDDAY('2004/1/1', -1) FROM RDBSDATABASE;  
          ADDDAY
```

```
=====
```

```
2003-12-31 00:00:00.0000
```

登録用スクリプト :

```
declare external function addDay  
timestamp, int  
returns timestamp  
entry_point 'addDay' module_name 'fbudf';
```

addWeek (引数 1, 引数 2)

引数 1 で指定された日付時刻型に、引数 2 で指定された週を加算して返します。

引数 1 : **TIMESTAMP** 型
引数 2 : **INTEGER** 型
戻り値 : **TIMESTAMP** 型

例 :

```
SQL> SELECT ADDWEEK('2004/1/1', 2) FROM RDBSDATABASE;  
ADDWEEK
```

```
=====  
2004-01-15 00:00:00.0000
```

登録用スクリプト :

```
declare external function addWeek  
timestamp, int  
returns timestamp  
entry_point 'addWeek' module_name 'fbudf';
```

addMnth (引数 1, 引数 2)

引数 1 で指定された日付時刻型に、引数 2 で指定された月を加算して返します。

引数 1 : **TIMESTAMP** 型
引数 2 : **INTEGER** 型
戻り値 : **TIMESTAMP** 型

例 :

```
SQL> SELECT ADDMONTH('2004/1/1', 2) FROM RDBSDATABASE;  
          ADDMONTH
```

```
=====
```

```
2004-03-01 00:00:00.0000
```

```
SQL> SELECT ADDMONTH('2004/1/1', 12) FROM RDBSDATABASE;  
          ADDMONTH
```

```
=====
```

```
2005-01-01 00:00:00.0000
```

登録用スクリプト :

```
declare external function addMnth  
timestamp, int  
returns timestamp  
entry_point 'addMnth' module_name 'fbudf';
```

addYear (引数 1, 引数 2)

引数 1 で指定された日付時刻型に、引数 2 で指定された年を加算して返します。

引数 1 : **TIMESTAMP** 型
引数 2 : **INTEGER** 型
戻り値 : **TIMESTAMP** 型

例 :

```
SQL> SELECT ADDYEAR('2004/1/1', 1) FROM RDBSDATABASE;  
        ADDYEAR
```

```
=====
```

2005-01-01 00:00:00.0000

登録用スクリプト :

```
declare external function addYear  
timestamp, int  
returns timestamp  
entry_point 'addYear' module_name 'fbudf';
```

addMlliSecond (引数 1, 引数 2)

引数 1 で指定された日付時刻型に、引数 2 で指定されたミリ秒を加算して返します。

引数 1 : **TIMESTAMP** 型
引数 2 : **INTEGER** 型
戻り値 : **TIMESTAMP** 型

例 :

```
SQL> SELECT ADDMLLISECOND('2004/1/1', 1) FROM RDBSDATABASE;  
      ADDMLLISECOND
```

```
=====
```

```
2004-01-01 00:00:00.0010
```

```
SQL> SELECT ADDMLLISECOND('2004/1/1', -1) FROM RDBSDATABASE;  
      ADDMLLISECOND
```

```
=====
```

```
2003-12-31 23:59:59.9990
```

登録用スクリプト :

```
declare external function addMlliSecond  
timestamp, int  
returns timestamp  
entry_point 'addMlliSecond' module_name 'fbudf';
```

addSecond (引数 1, 引数 2)

引数 1 で指定された日付時刻型に、引数 2 で指定された秒を加算して返します。

引数 1 : `TIMESTAMP` 型

引数 2 : `INTEGER` 型

戻り値 : `TIMESTAMP` 型

例 :

```
SQL> SELECT ADDSECOND('2004/1/1', 1) FROM RDBSDATABASE;  
ADDSECOND
```

```
=====
```

```
2004-01-01 00:00:01.0000
```

```
SQL> SELECT ADDSECOND('2004/1/1', -1) FROM RDBSDATABASE;  
ADDSECOND
```

```
=====
```

```
2003-12-31 23:59:59.0000
```

登録用スクリプト :

```
declare external function addSecond  
timestamp, int  
returns timestamp  
entry_point 'addSecond' module_name 'fbudf';
```

addMnute (引数 1, 引数 2)

引数 1 で指定された日付時刻型に、引数 2 で指定された分を加算して返します。

引数 1 : **TIMESTAMP** 型
引数 2 : **INTEGER** 型
戻り値 : **TIMESTAMP** 型

例 :

```
SQL> SELECT ADDMNUTE('2004/1/1', 1) FROM RDBSDATABASE;  
          ADDMNUTE
```

```
=====  
2004-01-01 00:01:00.0000
```

```
SQL> SELECT ADDMNUTE('2004/1/1', -1) FROM RDBSDATABASE;  
          ADDMNUTE
```

```
=====  
2003-12-31 23:59:00.0000
```

登録用スクリプト :

```
declare external function addMnute  
timestamp, int  
returns timestamp  
entry_point 'addMnute' module_name 'fbudf';
```

addHour (引数 1, 引数 2)

引数 1 で指定された日付時刻型に、引数 2 で指定された時間を加算して返します。

引数 1 : **TIMESTAMP** 型
引数 2 : **INTEGER** 型
戻り値 : **TIMESTAMP** 型

例 :

```
SQL> SELECT ADDHOUR('2004/1/1', 1) FROM RDBSDATABASE;  
ADDHOUR
```

```
=====  
2004-01-01 01:00:00.0000
```

```
SQL> SELECT ADDHOUR('2004/1/1', -1) FROM RDBSDATABASE;  
ADDHOUR
```

```
=====  
2003-12-31 23:00:00.0000
```

登録用スクリプト :

```
declare external function addHour  
timestamp, int  
returns timestamp  
entry_point 'addHour' module_name 'fbudf';
```

getExactTimestamp

現在の正確な **TimeStamp** を返します。この関数は他の **OS** に移植されるまでは **Win32** 環境でのみ動作します。

戻り値 : **TIMESTAMP** 型

例 :

```
SQL> SELECT GETEXACTTIMESTAMP() FROM RDBSDATABASE;  
      GETEXACTTIMESTAMP
```

```
=====  
2004-01-07 23:42:46.6980
```

登録用スクリプト :

```
declare external function getExactTimestamp  
timestamp returns parameter 1  
entry_point 'getExactTimestamp' module_name 'fbudf';
```

truncate (引数)

引数の小数部を切り捨てて整数部のみを返します。次に示す **I64TRUNCATE()** 関数と同じ関数を呼び出しています。**TRUNCATE()** 関数は **32** ビット精度となります。

引 数 : Integer
戻り値 : Integer

例 :

```
SQL> SELECT TRUNCATE(14.76) FROM RDBSDATABASE;  
TRUNCATE
```

```
=====
```

14

```
SQL> SELECT CAST(14.76 AS INTEGER) FROM RDBSDATABASE;  
CAST
```

```
=====
```

15

登録用スクリプト :

```
declare external function Truncate  
int by descriptor, int by descriptor  
returns parameter 2  
entry_point 'truncate' module_name 'fbudf';
```

※注意

TRUNCATE()関数に、Double Precision型・FLOAT型を渡すと戻り値はNULLになります。

```
SQL> CREATE TABLE T_DP (DP DOUBLE PRECISION);
```

```
SQL> INSERT INTO T_DP VALUES(14.76);
```

```
SQL> SELECT * FROM T_DP;
```

```
DP
```

```
=====
```

```
14.760000000000000
```

```
SQL> SELECT TRUNCATE(DP) FROM T_DP;
```

```
TRUNCATE
```

```
=====
```

```
<null>
```

```
SQL> CREATE TABLE T_NM (NM NUMERIC(18,4));
```

```
SQL> INSERT INTO T_NM VALUES (14.72);
```

```
SQL> INSERT INTO T_NM VALUES (14.5);
```

```
SQL> INSERT INTO T_NM VALUES (14.4999);
```

```
SQL> SELECT TRUNCATE(NM) FROM T_NM;
```

```
TRUNCATE
```

```
=====
```

```
14
```

```
14
```

```
14
```

i64Truncate (引数)

引数の小数部を切り捨てて整数部のみを返します。次に示す `TRUNCATE()` 関数と同じ関数を呼び出しています。`I64TRUNCATE()` 関数は **64** ビット精度となります。

引 数 : `NUMERIC(18)`

戻り値 : `NUMERIC(18)`

例 :

```
SQL> SELECT I64TRUNCATE(14.76) FROM RDBSDATABASE;  
          I64TRUNCATE
```

```
=====
```

```
          14
```

登録用スクリプト :

```
declare external function i64Truncate  
numeric(18) by descriptor, numeric(18) by descriptor  
returns parameter 2  
entry_point 'truncate' module_name 'fbudf';
```

※注意

TRUNCATE()関数と同様、Double Precision型・FLOAT型を渡すと戻り値はNULLになります。

```
SQL> CREATE TABLE T_DP (DP DOUBLE PRECISION);
```

```
SQL> INSERT INTO T_DP VALUES(14.76);
```

```
SQL> SELECT I64ROUND(DP) FROM T_DP;
```

```
ROUND
```

```
=====
```

```
<null>
```

```
SQL> CREATE TABLE T_NM (NM NUMERIC(18,4));
```

```
SQL> INSERT INTO T_NM VALUES (14.72);
```

```
SQL> INSERT INTO T_NM VALUES (14.5);
```

```
SQL> INSERT INTO T_NM VALUES (14.4999);
```

```
SQL> SELECT I64TRUNCATE(NM) FROM T_NM
```

```
I64TRUNCATE
```

```
=====
```

```
14
```

```
14
```

```
14
```

round (引数)

引数を四捨五入して返します。

引数 : Integer

戻り値 : Integer

例 :

```
SQL> SELECT ROUND(14.5) FROM RDBSDATABASE;
```

```
      ROUND  
=====
```

15

```
SQL> SELECT ROUND(14.4999) FROM RDBSDATABASE;
```

```
      ROUND  
=====
```

14

登録用スクリプト :

```
declare external function Round  
int by descriptor, int by descriptor  
returns parameter 2  
entry_point 'round' module_name 'fbudf';
```

※注意

Double Precision型・FLOAT型を渡すと戻り値はNULLになります。

```
SQL> SELECT ROUND(DP) FROM T_DP;
      ROUND
=====
      <null>
```

小数部を必要とする場合はNumeric型を使用して下さい。

```
SQL> CREATE TABLE T_NM (NM NUMERIC(18,4));
SQL> INSERT INTO T_NM VALUES (14.72);
SQL> INSERT INTO T_NM VALUES (14.5);
SQL> INSERT INTO T_NM VALUES (14.4999);
SQL> SELECT ROUND(NM) FROM T_NM;
      ROUND
=====
      15
      15
      14
```

i64Round (引数)

引数を四捨五入して返します。

引数 : `Numeric(18, 4)`
戻り値 : `Numeric(18, 4)`

例 :

```
SQL> SELECT ROUND(14.5) FROM RDBSDATABASE;  
      ROUND  
=====  
      15
```

```
SQL> SELECT ROUND(14.4999) FROM RDBSDATABASE;  
      ROUND  
=====  
      14
```

登録用スクリプト :

```
declare external function i64Round  
numeric(18, 4) by descriptor, numeric(18, 4) by descriptor  
returns parameter 2  
entry_point 'round' module_name 'fbudf';
```

※注意

Double Precision型・FLOAT型を渡すと戻り値はNULLになります。

```
SQL> SELECT I64ROUND(DP) FROM T_DP;  
ROUND  
=====  
<null>
```

小数部を必要とする場合はNumeric型を使用して下さい。

```
SQL> CREATE TABLE T_NM (NM NUMERIC(18,4));  
SQL> INSERT INTO T_NM VALUES (14.72);  
SQL> INSERT INTO T_NM VALUES (14.5);  
SQL> INSERT INTO T_NM VALUES (14.4999);  
SQL> SELECT I64ROUND(NM) FROM T_NM  
I64ROUND  
=====  
15.0000  
15.0000  
14.0000
```

dpower (引数 1, 引数 2)

引数 1 を引数 2 乗して返します。

引数 1, 2 : Double Precision

戻り値 : Double Precision

例 :

```
SQL> SELECT DPOWER(2, 3) FROM RDBSDATABASE;  
          DPOWER
```

```
=====
```

8.000000000000000

登録用スクリプト :

```
declare external function dPower  
double precision by descriptor, double precision by descriptor,  
double precision by descriptor  
returns parameter 3  
entry_point 'power' module_name 'fbudf';
```

string2blob (引数)

文字列を **BLOB** に変換して返します。

引 数 : VarChar(300)
戻り値 : BLOB

例 :

```
SQL> CREATE TABLE T_BLOB (BSTR BLOB SUB_TYPE 1);
SQL> INSERT INTO T_BLOB VALUES (STRING2BLOB('TEST'));
SQL> INSERT INTO T_BLOB VALUES (STRING2BLOB('あいう'));
SQL> SELECT * FROM T_BLOB;
```

BSTR

```
=====
a8:0
```

BSTR:

```
あいう
=====
```

※マルチバイトも通りました。

登録用スクリプト :

```
declare external function string2blob
varchar(300) by descriptor,
blob returns parameter 2
entry_point 'string2blob' module_name 'fbudf';
```

フリーソフト・オープンソース UDF 編

Firebird/InterBase の UDF は、手軽に作成できることもあり、各種のライブラリがフリー・ウェアやオープンソース・プロダクトとして提供されています。IBPhoenix 社のサイトで紹介されているものを以下に示します。

名称：FUDlibrary

概要：60 を超えるの一般的な UDF 関数を含んでいます

作者：David Bowyer

配布：Open Source (GPL)

名称：MER Systems free UDFlib for InterBase V6.0 (Linux)

概要：Linux 版の InterBase V6/Firebird 向け FreeUDFlib(Win32 版向けのバグフィックスを含みません)です。Rbert Shcieck が移植しました。Dialect 1 のみで使用できます。本ライブラリによってトリガ、ストアド、SQL 文で 60 を超える関数が使用出来るようになります。

いくつかの BLOB 向け UDF は機能しません。

このライブラリ (Linux 版のみ) は、Dialect3 データベースで動作しません(2001/6/22 現在)

作者：Rob Schieck

配布：Freeware

名称：FreeUDFlib - Free ODS 9 UDFs for Windows

作者：By Greg Deatz

配布：Freeware

名称：Free UDFlib for InterBase V6 for Windows

概要：Claudio Valderrama が Greg Deatz' が作成した IB6.0.x/Firebird win32 版向け free UDFlib を拡張し、バグフィックスをしたものです。

いくつかの関数は機能していませんでした。それらは、f_BlobAsPchar、f_BlobRight、f_Character ですが、きちんとした値を例外無しで返すようになりました。また、f_BlobMid にも多少の改良がおこなれています。

デバッグ版では、デバッグメッセージが多少追加・修正されています。

また、Mers リストの誰かが Blob のバイナリコンペアを所望したので、新しく f_Blobcmp を追加しましたが、まだ小さな Blob でのテストしかされていません。これまでのところ、変更は Win32 版のみに行われています。

作者：Gregory Deatz and others (Community Effort)

配布：Freeware

名称：FreeUDFlibC for UNIX and Linux

作者：Greg Deatz

配布：Freeware

名称：BlobUDFlib V1.0.0.9

概要：InterBase の Blob を対象としたライブラリ。

作者：Kevin Burge

配布：Freeware

名称：Blob UDFs

概要：blob アクセスのための UDF 集。

作者：Ian Newby

配布：Freeware

名称：FreePascal UDFs

概要：Free Pascal 向け UDF ライブラリ。このライブラリが、Delphi で書かれた UDF を Linux へ移植する出発点になることを期待しています。

作者：Frank Schlottmann-Gödde

配布：Freeware

名称：xLibUDF V1.2

概要：string、mutex、GUID の生成、ストアド・トリガのデバッグ用に作成された UDF ライブラリ。

作者：SoftLab MIL-TEC Ltd

配布：Freeware

名称：External File UDF's V1.0

概要：Firebird/InterBase(Windows のみ)向けの外部関数で、外部ファイルを扱うための基本的な機能を提供します。ファイルの作成、書込、削除、コピー、マージなどです。データ・インプットに関するログファイルを外部に作成することや、データベースから静的な HTML ページを生成するのに役立ちます。

作者：David Revill

配布：Freeware

名称：SR UDFs for InterBase & Firebird V1.02

概要：SR UDFs ライブラリはパワフルな UDF 関数群です(約 60 以上)。Firebird と InterBase におけるデータ操作の可能性を広げます。

作者：Safa Rimeh

配布 : Freeware

名称 : TBUDF Library

概要 : UDF ライブラリを Delphi・Kylix へ移植したもの。この中には、Claudio Valderama 作の関数 (NVL、ISNULL 等) にその他の便利な関数をいくつか追加してあります。

NULL と dow/sdow 関数は 2002/7/3 に Anthony J. Caduto によって追加されました。

オリジナルの Null 関数は Claudio Valderama 氏によるものです。

スペシャル・サンクス :

Ann Harrison 氏は descriptor を通して varchar、cstring、text の各フォーマットを提供してくれました。

Rudy Velthuis:string ルーチンでのバッファ処理で手助けしてくれました。

Henner Kollman:setnull・isnull 関数を移植してくれました。

そして、Firebird-devel ニュースグループで助けてくれた全ての皆さん。

作者 : Tony Caduto

配布 : Open Source (MPL)

名称 : rFunc UDF Library V2.01

概要 : string、bit、数学関数についての UDF ライブラリ、その他 date、time と Blob についての操作も含んでいます。また、パーサー (計算式表現) についても含んでいます。

InterBase 4.3、5.x、6.0 (Windows 9x、NT、2k) をサポートし、InterBase 5.x、6.x (Linux) を、また、Firebird 1.x をサポートします。

ライブラリは 90 以上の関数と疑似関数が含まれています。

ライブラリは C++ で書かれ、ソースコードと共に配布されています。

作者 : Polaris Software

配布 : Open Source (LGPL)

名称 : uuidUDF

概要 : このライブラリはソースコードとコンパイル済みのバイナリを含んでいます。ライブラリは GUID の生成に関するものです。コンパイル済みバイナリは Windows と Linux 用となっています。Windows の dll は gcc のクロスプラットフォームコンパイラでコンパイルされています。

含まれる関数は以下のものです :

create_guid() は 36 キャラクタの GUID 文字列を生成します。

create_uuid() は 22 キャラクタの GUID 文字列を生成します。これは Firebird のインデックスでプリフィクスの圧縮が出来るようにセクションのオーダーが逆転されたものです。uuid で使用される全てのキャラクタは URL で有効なキャラクタです。

guid_to_uuid(guid char 36) uuid を生成するために、guid を逆転して圧縮します。

uuid_to_guid(uuid char 22) uuid を guid にコンバートします。

作者 : Ian Newby

配布 : Freeware

XLibUDF

XLibUDF は、SoftLab ML-TEC Ltd が作成し、配布していたフリーソフトの UDF ライブラリです。Windows 版のみが存在し、ソースコードは公開されていません。

SoftLab は、現在では SoftComplete と名称を変えており、XLibUDF の配布は中止されてしまっているようです。

ただし、オリジナルのライセンスは BSD ライクなもので、再配布は自由である旨記載されているため、筆者のホームページで配布しております。

<http://homepage2.nifty.com/tonneko/>

XLibUDF には 13 の UDF が含まれていますが、このうち、SubStr() 関数・StrLength() 関数は、ib_UDF ライブラリに含まれている SubStrLen() 関数・StrLen() 関数と同様ですので、説明を省略します。また、SubStr() 関数の名称が ib_UDF ライブラリの SubStr() 関数と同じなので両者を混同しないようにして下さい。

また、WaitMtx() 関数・FreeMtx() 関数は動作が確認できなかったため、説明を省略しています。

StrTrim(引数)

指定された ASCII 文字列の両側に存在する空白を取り除きます。

登録用スクリプトでは、引数と戻り値が **CSTRING(255)** となっていますが、255 文字までに制限されているわけではありません。実際は、32,767 文字の **Firebird** の文字列の制限まで利用することが可能です。その場合、登録用スクリプトの文字列長指定を必要な数値に変更して登録して下さい。

引 数 : CString(255)

戻り値 : CString(255)

例 :

```
SQL> SELECT STRTRIM(' ABC ') FROM RDBSDATABASE;
```

```
STRTRIM
```

```
=====
```

```
ABC
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION StrTrim
```

```
CSTRING(255)
```

```
RETURNS CSTRING(255) FREE_IT
```

```
ENTRY_POINT 'StrTrim' MDULE_NAME 'XLIBUDF.DLL';
```

StrUCase(引数)

指定された ASCII 文字列を全て大文字に変換します。
登録用スクリプトでは、引数と戻り値が **CSTRING(255)** となっていますが、255 文字までに制限されているわけではありません。実際は、32,767 文字の **Firebird** の文字列の制限まで利用することが可能です。
その場合、登録用スクリプトの文字列長指定を必要な数値に変更して登録して下さい

引 数 : CString(255)

戻り値 : CString(255)

例 :

```
SQL> SELECT STRUCASE('abc') FROM RDBSDATABASE;
```

```
STRUCASE
```

```
=====
```

```
ABC
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION StrUCase
```

```
CSTRING(255)
```

```
RETURNS CSTRING(255)
```

```
ENTRY_POINT 'StrUCase' MODULE_NAME 'XLIBUDF.DLL';
```

StrLCase(引数)

指定された ASCII 文字列を全て小文字に変換します。

登録用スクリプトでは、引数と戻り値が **CSTRING(255)** となっていますが、255 文字までに制限されているわけではありません。実際は、**32,767** 文字の **Firebird** の文字列の制限まで利用することが可能です。

その場合、登録用スクリプトの文字列長指定を必要な数値に変更して登録して下さい

引 数 : CString(255)

戻り値 : CString(255)

例 :

```
SQL> SELECT STRUCASE('ABC') FROM RDBSDATABASE;
```

```
STRUCASE
```

```
=====
```

```
abc
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION StrLCase
```

```
CSTRING(255)
```

```
RETURNS CSTRING(255)
```

```
ENTRY_POINT 'StrLCase' MODULE_NAME 'XLIBUDF.DLL';
```

StrFirst(引数 1, 引数 2)

指定された ASCII 文字列の先頭から指定された文字数を返します。

登録用スクリプトでは、引数と戻り値が **CSTRING(255)** となっていますが、255 文字までに制限されているわけではありません。実際は、32,767 文字の **Firebird** の文字列の制限まで利用することが可能です。

その場合、登録用スクリプトの文字列長指定を必要な数値に変更して登録して下さい

引 数 : **CString(255)**

戻り値 : **INTEGER**

例 :

```
SQL> SELECT STRFIRST('ABCDEFG', 3) FROM RDBSDATABASE;
```

```
STRFIRST
```

```
=====
```

```
ABC
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION StrFirst
```

```
CSTRING(255), INTEGER
```

```
RETURNS CSTRING(255)
```

```
ENTRY_POINT "StrFirst" MODULE_NAME "XLIBUDF.DLL";
```

StrLast(引数 1, 引数 2)

指定された ASCII 文字列の最後から指定された文字数を返します。

登録用スクリプトでは、引数と戻り値が **CSTRING(255)** となっていますが、255 文字までに制限されているわけではありません。実際は、32,767 文字の **Firebird** の文字列の制限まで利用することが可能です。

その場合、登録用スクリプトの文字列長指定を必要な数値に変更して登録して下さい

引 数 : **CString(255)**

戻り値 : **INTEGER**

例 :

```
SQL> SELECT STRLAST(' ABCDEFG', 3) FROM RDBSDATABASE;
```

```
STRFIRST
```

```
=====
```

```
EFG
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION StrLast
```

```
CSTRING(255), INTEGER
```

```
RETURNS CSTRING(255)
```

```
ENTRY_POINT 'StrLast' MODULE_NAME 'XLIBUDF.DLL';
```

ANSILike(引数 1, 引数 2)

引数 1 で指定された ASCII 文字列内に、引数 2 で指定された ASCII 文字列が存在する場合、1 を返します。大文字小文字は区別されず、一致する文字列が存在しない場合は 0 を返します。

登録用スクリプトでは、引数と戻り値が CSTRING(255) となっていますが、255 文字までに制限されているわけではありません。実際は、32,767 文字の Firebird の文字列の制限まで利用することが可能です。

その場合、登録用スクリプトの文字列長指定を必要な数値に変更して登録して下さい

引 数 : CString(255)

戻り値 : CString(255)

例 :

```
SQL> SELECT ANSILIKE('ABCDEF', 'BC') FROM RDBSDATABASE;
ANSILIKE
=====
1
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION ANSILike
CSTRING(255), CSTRING(255)
RETURNS INTEGER BY VALUE
ENTRY_POINT 'ANSILike' MODULE_NAME 'XLIBUDF.DLL';
```

CreateUID

GUID 文字列を生成して返します。GUID 文字列は文字列として返されますが、「-」は付加されません。

引 数 : 無し

戻り値 : CString(32)

例 :

```
SQL> SELECT CREATEUID() FROM RDBSDATABASE;
CREATEUID
=====
C57BBFEA2E043C1AD74012AFD5C83ED
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION CreateUID
RETURNS CSTRING(32) FREE_IT
ENTRY_POINT 'CreateUID' MODULE_NAME 'XLIBUDF.DLL';
```

※注意

CreateUID 関数が返す文字列を検証すると、32 文字ではなく、31 文字となっています。これでいいのか疑問です。

WriteDebug(引数)

WriteDebug 関数は、ストアドプロシージャやトリガーのデバッグ用に利用することを想定したもので、引数で指定された文字列を **IBDebugWnd.exe** へと表示します。

IBDebugWnd.exe は、**WriteDebug()** 関数を実行する前に最低 1 回は直接実行しておかなくてはなりません。初回実行時に、自分自身を COM オブジェクトとしてシステムに登録し、**WriteDebug** 関数からの呼出が可能となります。

引 数 : **CString(255)**

戻り値 : 無し

例 :

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION WriteDebug
CSTRING(255)
RETURNS INTEGER BY VALUE
ENTRY_POINT 'WriteDebug' MDDULE_NAME 'XLIBUDF.DLL';
```

TimeStamp()

現在の時刻を OS タイムスタンプ型で返します。

Firebird では、**TimeStamp** は予約語となっているため、**DECLARE EXTERNAL FUNCTION** 文で、**XTIMESTAMP** 等として登録する必要があります。

引 数：無し

戻り値：INTEGER

例：

登録用スクリプト：

```
DECLARE EXTERNAL FUNCTION xTimeStamp  
RETURNS INTEGER BY VALUE  
ENTRY_POINT 'TimeStamp' MODULE_NAME 'XLIBUDF.DLL';
```

uui dUDF

uui dUDF は、Ian Newby 氏によってコントリビュートされたライブラリです。ソースコードも含まれていますが、IBPhoenix 社のサイトでは、Freeware に分類されています。実際にソースコードを見る限りでは、単に「AS IS」であるという以外のライセンス規定がないので、オープンソースとは言えないということなのでしょう。

<http://www.ibphoenix.com/downloads/uuidlib.zip>

こちらからダウンロードすることが出来ます。Linux 版と Windows 版のコンパイル済バイナリが含まれています。本ライブラリで取り扱う GUID 文字列と UUID 文字列の違いは、36 文字の GUID 文字列を 22 文字に圧縮し、かつ Firebird のインデックスで圧縮可能な形式にするということですが、具体的には以下ようになります。

GUID : 0d5c75e0-7b0e-11d8-ad6b-c0b52712533d
UUID : 1!hGQGInpfff5MBu!BL5LU

GUID_CREATE

GUID_CREATE()関数は、36文字のGUID文字列を返します。文字列は、xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx と「-」で区切られた形式となります。

引 数：無し

戻り値：CString(36)

例：

```
SQL> SELECT GUID_CREATE() FROM RDBSDATABASE;
```

```
GUID_CREATE
```

```
=====
0d5c75e0-7b0e-11d8-ad6b-c0b52712533d
```

登録用スクリプト：

```
DECLARE EXTERNAL FUNCTION GUID_CREATE
```

```
  CSTRING(36)
```

```
  RETURNS PARAMETER 1
```

```
  ENTRY_POINT 'fn_guid_create' MODULE_NAME 'uuidlib';
```

GUID_TO_UUID (引数)

GUID_TO_UUID() 関数は、GUID_CREATE() 関数で生成された GUID 文字列を UUID 文字列に変換します。

引 数 : CString(36)

戻り値 : CString(22)

例 :

```
SQL> SELECT GUID_TO_UUID('0d5c75e0-7b0e-11d8-ad6b-c0b52712533d') FROM RDBSDATABASE;
```

```
GUID_TO_UUID
```

```
=====
1!hQGInpffF5Bu!BL5LU
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION GUID_TO_UUID
```

```
  CString(36),
```

```
  CString(22)
```

```
  RETURNS PARAMETER 2
```

```
  ENTRY_POINT 'fn_squash_guid' MDULE_NAME 'uuidlib';
```

UUID_CREATE

UUID_CREATE()関数は、22文字のUUID文字列を返します。

引数：無し
戻り値：CString(22)

例：

```
SQL> SELECT UUID_CREATE() FROM RDBSDATABASE;  
UUID_CREATE
```

```
=====
```

```
1!hgQGInpffF5MSjDIW!
```

登録用スクリプト：

```
DECLARE EXTERNAL FUNCTION UUID_CREATE  
  CSTRING(22)  
  RETURNS PARAMETER 1  
  ENTRY_POINT 'fn_uuid_create' MODULE_NAME 'uuidlib';
```

UUID_TO_GUID(引数)

UUID_TO_GUID() 関数は、UUID_CREATE() 関数で生成された UUID 文字列を GUID 文字列に変換します。

引 数 : CString(22)
戻り値 : CString(36)

例 :

```
SQL> SELECT UUID_TO_GUID('1!hGQGImpffF5MRS!BL5LU') FROM RDBSDATABASE;  
UUID_TO_GUID  
=====  
0d5c75e0-75e0-11d8-ad6b-c0b52712533d
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION UUID_TO_GUID  
    CString(22),  
    CString(36)  
    RETURNS PARAMETER 2  
ENTRY_POINT 'fn_expand_uuid' MODULE_NAME 'uuiplib';
```

External File UDF's

External File UDF's は、David Reville 氏によってコントリビュートされた UDF ライブラリです。ライセンス携帯は、ZIP License と BSD ライセンスを合わせたようなものになっています。ソースコードが含まれていないため、Freeware という分類とされています。

<http://www.ibphoenix.com/downloads/extfileudf.zip>

こちらからダウンロードすることが出来ます。Windows 版のみとなっています。

また、UDF のみでなく、ライブラリに含まれる UDF を利用して File の生成や削除、HTML ファイルの生成などを行うストアドプロシージャのサンプルも付属しています。

本ライブラリ中で、ファイル名を示すパス文字列を指定する場合には、区切り文字として「\」ではなく「/」を使用する必要があります。「\」を使用した場合、全角の「\」（バックslash）として扱われますので注意して下さい。

さて、本ライブラリの特徴であるファイル関連の関数は、整数型の戻り値を返します。それぞれ以下のような意味を持っています。

0	Success
2	File not found.
3	Invalid file name.
4	Too many open files.
5	Access denied.
100	EOF.
101	Disk full.
106	Invalid input.
-1	Other Error

NoNull (引数)

`NoNull()` 関数は、引数で指定された文字列が `Null` である場合に'' (空文字) を返します。文字列の結合などで、`Null` が一つでも含まれていると全体の戻り値が `Null` になってしまうため、この関数を使うとそうした不具合を回避することができます。

引 数 : `CString(1024)`

戻り値 : `CString(1024)`

例 :

```
SQL> SELECT RDB$DESCRIPTION FROM RDB$DATABASE;  
RDB$DESCRIPTION
```

```
=====
```

```
<null>
```

```
SQL> SELECT NONULL(RDB$DESCRIPTION) FROM RDB$DATABASE;
```

```
NONULL
```

```
=====
```

登録用スクリプト :

```
declare external function Nonull  
  cstring(1024)  
  returns cstring(1024) FREE_IT  
  entry_point 'Nonull' module_name 'udf_file.dll';
```

File_Write(引数 1, 引数 2)

File_Write() 関数は、引数 2 で指定された文字列を引数 1 で指定されたファイルへ書き込みます。
指定されたファイルが存在しない場合は、新しくファイルを作成し、既に存在する場合は既存のファイルに追記します。

引数 1 : CString(1024)
引数 2 : CString(1024)
戻り値 : INTEGER

例 :

```
SQL> SELECT FILE_WRITE('C:/TEST.TXT', 'ABC') FROM RDB$DATABASE;  
FILE_WRITE  
=====  
0
```

登録用スクリプト :

```
declare external function File_write  
cstring(1024), cstring(1024)  
returns integer by value  
entry_point 'File_write' module_name 'udf_file.dll';
```

File_WriteLn(引数 1, 引数 2)

File_WriteLn() 関数は、引数 2 で指定された文字列を引数 1 で指定されたファイルへ書き込みます。その際に、CRLF を追加します。

指定されたファイルが存在しない場合は、新しくファイルを作成し、既に存在する場合は既存のファイルに追記します。

引数 1 : CString(1024)

引数 2 : CString(1024)

戻り値 : INTEGER

例 :

```
SQL> SELECT FILE_WRITELN('C:/TEST.TXT', 'ABC') FROM RDBSDATABASE;
```

```
FILE_WRITE
=====
          0
```

登録用スクリプト :

```
declare external function File_writeLn
cstring(1024),cstring(1024)
returns integer by value
entry_point 'File_writeLn' module_name 'udf_file.dll';
```

File_delete(引数)

`File_delete()`関数は、引数で指定されたファイルを削除します。

引数: `CString(1024)`

戻り値: `INTEGER`

例:

```
SQL> SELECT FILE_DELETE('C:/TEST.TXT') FROM RDBSDATABASE;  
FILE_DELETE
```

```
=====
```

```
0
```

登録用スクリプト:

```
declare external function File_delete  
  cstring(1024)  
  returns integer by value  
  entry_point 'File_delete' module_name 'udf_file.dll';
```

File_create(引数)

`File_create()`関数は、引数で指定されたファイル名の空のファイルを作成します。同じ名称で既存のファイルが存在する場合、そのファイルは上書きされます。

引 数 : CString(1024)

戻り値 : INTEGER

例 :

```
SQL> SELECT FILE_CREATE('C:/TEST.TXT') FROM RDBSDATABASE;  
FILE_CREATE
```

```
=====
```

```
0
```

登録用スクリプト :

```
declare external function File_create  
cstring(1024)  
returns integer by value  
entry_point 'File_create' module_name 'udf_file.dll';
```

Strformat(引数 1, 引数 2, 引数 3)

Strformat() 関数は、引数 1 で指定された置換文字列を含む文字列を、引数 3 で指定された区切り文字で区切られた、引数 2 の文字列を使用して書式化します。

引数 1 : CString(1024)

引数 2 : CString(1024)

引数 3 : CString(1)

戻り値 : CString(1024)

例 :

```
SQL> SELECT STRFORMAT(' FIELD1= % FIELD2= %', ' 123,456', ',') FROM RDBSDATABASE;
```

```
STRFORMAT
```

```
=====
FIELD1= 123 FIELD2= 456
```

登録用スクリプト :

```
declare external function Strformat
cstring(1024), cstring(1024), cstring(1)
returns Cstring(1024) FREE_IT
entry_point 'Strformat' module_name 'udf_file.dll';
```

File_rename(引数 1, 引数 2)

`File_rename()`関数は、引数 1 で指定されたファイルの名前を、引数 2 で指定された名前に変更します。

引数 1 : `CString(1024)`

引数 2 : `CString(1024)`

戻り値 : `INTEGER`

例 :

登録用スクリプト :

```
declare external function File_rename
cstring(1024), cstring(1024)
returns integer by value
entry_point 'File_rename' module_name 'udf_file.dll';
```

File_concat (引数 1, 引数 2)

`File_concat()` 関数は、引数 1 で指定されたカンマ区切りの文字列で示される複数のファイルを結合して、引数 2 で指定されたファイルを作成します。

引数 1 : CString(1024)

引数 2 : CString(1024)

戻り値 : INTEGER

例 :

```
SQL> SELECT FILE_CONCAT('C:/TEST.TXT,C:/TEST2.TXT','C:/TEST3.TXT') FROM RDBSDATA
```

```
BASE;
```

```
FILE_CONCAT
```

```
=====
```

```
0
```

登録用スクリプト :

```
declare external function File_concat  
cstring(1024), cstring(124)  
returns integer by value  
entry_point 'File_concat' module_name 'udf_file.dll';
```

Padleft(引数 1, 引数 2)

Padleft() 関数は、引数 1 で指定された文字列が、引数 2 で指定された長さになるよう、文字列の左側を半角スペースで埋めて返します。引数 1 で指定された文字列が引数 2 で指定された長さ以上の場合は何も行いません。

引数 1 : CString(1024)

引数 2 : INTEGER

戻り値 : CString(1024)

例 :

```
SQL> SELECT PADLEFT(' abc', 5) FROM RDBSDATABASE;
```

```
PADLEFT
```

```
=====
```

```
 abc
```

登録用スクリプト :

```
declare external function Padleft
cstring(1024), integer
returns cstring(1024) FREE_IT
entry_point 'Padleft' module_name 'udf_file.dll';
```

Padright (引数 1, 引数 2)

Padright()関数は、引数 1 で指定された文字列が、引数 2 で指定された長さになるよう、文字列の右側を半角スペースで埋めて返します。引数 1 で指定された文字列が引数 2 で指定された長さ以上の場合は何も行いません。

引数 1 : CString(1024)

引数 2 : INTEGER

戻り値 : CString(1024)

例 :

```
SQL> SELECT PADRIGHT('abc', 5) FROM RDBSDATABASE;
```

```
PADLEFT
```

```
=====
```

```
abc
```

登録用スクリプト :

```
declare external function Padright
```

```
cstring(1024), integer
```

```
returns cstring(1024) FREE_IT
```

```
entry_point 'Padright' module_name 'udf_file.dll';
```

Squotedstr(引数)

`Squotedstr()`関数は、引数で指定された文字列を、一重引用符で囲んで返します。

引 数 : `CString(1024)`

戻り値 : `CString(1024)`

例 :

```
SQL> SELECT SQUOTEDSTR('abc') FROM RDBSDATABASE;
```

```
SQUOTEDSTR
```

```
=====
```

```
'abc'
```

登録用スクリプト :

```
declare external function Squotedstr
```

```
cstring(1024)
```

```
returns cstring(1024) FREE_IT
```

```
entry_point 'Squotedstr' module_name 'udf_file.dll';
```

dquotedstr(引数)

`dquotedstr()`関数は、引数で指定された文字列を二重引用符で囲んで返します。

引数 : CString(1024)

戻り値 : CString(1024)

例 :

```
SQL> SELECT DQUOTEDSTR(' abc') FROM RDBSDATABASE;
```

```
DQUOTEDSTR
```

```
=====
```

```
"abc"
```

登録用スクリプト :

```
declare external function dquotedstr
```

```
cstring(1024)
```

```
returns cstring(1024) FREE_IT
```

```
entry_point 'Dquotedstr' module_name 'udf_file.dll';
```

datetimeformat(引数1, 引数2)

`datetimeformat()`関数は、引数2で指定された日付時刻型の値を、引数1で指定された書式に従って形式化します。

引数1 : CString(1024)

引数2 : TIMESTAMP

戻り値 : CString(1024)

例 :

```
SQL> SELECT DATETIMEFORMAT('yyyy/mm/dd', CURRENT_TIMESTAMP) FROM RDBSDATABASE;  
DATETIMEFORMAT
```

```
=====
```

```
2004/03/22
```

```
SQL> SELECT DATETIMEFORMAT('yyyy/mm/dd hh:nn:ss', CURRENT_TIMESTAMP) FROM RDBSDATABASE;  
DATETIMEFORMAT
```

```
=====
```

```
2004/03/22 22:10:11
```

登録用スクリプト :

```
declare external function Datetimeformat  
cstring(1024), timestamp  
returns cstring(1024) FREE_IT  
entry_point 'Datetimeformat' module_name 'udf_file.dll';
```

Numberformat(引数 1, 引数 2)

Numberformat()関数は、引数 2 で指定された数値型の値を、引数 1 で指定された書式に従って形式化します。

引数 1 : CString(1024)

引数 2 : CString(1024)

戻り値 : CString(1024)

例 :

```
SQL> select numberformat('0.00', 2.1) from rdb$database;
```

```
NUMBERFORMAT
```

```
=====
```

```
2.10
```

登録用スクリプト :

```
declare external function Numberformat
```

```
cstring(1024), cstring(1024)
```

```
returns cstring(1024) FREE_IT
```

```
entry_point 'Numberformat' module_name 'udf_file.dll';
```

External File UDF's 付属ストアプロシージャ

External File UDF's ライブラリには、UDF を活用するためのストアプロシージャが付属しています。

以下の 4 つのストアプロシージャは、単にそれぞれの関数をラップしたものです。使い方はそれぞれの関数と同様となります。

本書の例題で使用しているように、`rdb$database` を利用した 1 行テーブルテクニックを利用すれば、こうしたストアを使用しなくても UDF を有効に活用することも可能です。

`Fileconcat(sourcefiles varchar(1024), destfile varchar(255))` returns (res integer)

`Filedelete(filename varchar(255))` returns (res integer)

`Filecreate(filename varchar(255))` returns (res integer)

`Filename(oldfilename varchar(255), newfilename varchar(255))` returns (res integer)

さて、同様に External File UDF's ライブラリに付属してくる DUMP_ASHTML プロシージャは、データベースに存在するテーブルの一覧と、ストアードプロシージャの一覧を HTML ファイルへ書き出すというものです。

引数にファイル名を取りますので、以下のような使い方となります。

例 :

```
SQL> EXECUTE PROCEDURE DUMP_ASHTML('d:/test.html');
      RES
=====
      0
```

登録用スクリプト :

```
/* example use of functions to Dump some system info to HTML File */
```

```
COMMIT WORK;
SET AUTODDL OFF;
SET TERM ^ ;
```

```
/* Stored procedures */
```

```
CREATE PROCEDURE DUMP_ASHTML
(
  AFILENAME VARCHAR(255)
)
RETURNS
(
  RES INTEGER
)
AS
BEGIN EXIT; END ^
```

```
ALTER PROCEDURE DUMP_ASHTML
```

```
(
  AFILENAME VARCHAR(255)
)
```

```
RETURNS
```

```
(
  RES INTEGER
)
```

```
AS
```

```
declare variable pinputs integer;
```

```
declare variable poutputs integer;
```

```
declare variable pname varchar(32);
```

```
begin
```

```
  res=file_create(afilename);
```

```
  if (res<>0) then
```

```
    begin
```

```
      suspend;
```

```
      exit;
```

```
    end
```

```
  res=file_writeln(afilename, '<h1>Tables and Views</h1><br><TABLE border=1><TR><th>Table Name</th></TR>');
```

```
  if (res<>0) then
```

```
    begin
```

```
      suspend;
```

```
      exit;
```

```
    end
```

```
for
```

```
  select RdbSRelation_name from rdb$relations where rdb$system_flag=0 order by RdbSRelation_name into :pname do
```

```
  begin
```

```
    res=file_writeln(afilename, '<TR><td>' || nullif(pname, '') || '</td><td>');
```

```
    if (res<>0) then
```

```
      begin
```

```
        suspend;
```

```
        exit;
```

```
      end
```

```
    end
```

```
  res=file_writeln(afilename, '</TABLE>');
```

```
/* Procedures */
```

```
  res=file_writeln(afilename, '<h1>Stored Procedures</h1><br><TABLE border=1> <TR><th>Procedure Name</th><th>Inputs</th><th>Outputs</th></TR>');
```

```

if (res<>0) then
  begin
    suspend;
    exit;
  end
for
select rdb$Procedure_name , rdb$procedure_inputs, rdb$procedure_outputs from rdb$Procedures
order by rdb$Procedure_name into pname, pinputs, poutputs
do
  begin
    res=file_writeln(filename, strformat(' <TR><td>% </td><td align="right">% </td><td align="right">%
</td></TR>', nonull (pname) || ', ' || nonull (pinputs) || ', ' || nonull (poutputs) || ' ', ', '));
    if (res<>0) then
      begin
        suspend;
        exit;
      end
    end
    res=file_writeln(filename, ' </TABLE>');
    suspend;
  end
end
^

SET TERM ; ^
COMMIT WORK;
SET AUTODDL ON;

```

UDF 作成編

さて、ここまでは既に存在する UDF ライブラリの利用法を解説してきましたが、次は独自に UDF を作成してみたいと思います。UDF を作成するにあたっては、筆者が普段利用している環境である Delphi 6 を使用しています。また、必要に応じて C 言語のコードも示しています。

UDF 作成の基礎

まず、Firebird RDBMS 傾向と対策 3 でご紹介した IB_UDF に含まれる RAND() 関数が、連続した呼出に対して同じ乱数値を返してしまう問題について、オリジナルのコードから追って見たいと思います。

以下に示すのは、firebird-1.5.0.4290.tar.bz2 を展開して、src\extlib に含まれている ib_udf.c のソースコードの一部です。大変簡潔なので、UDF 作成が手軽にできることをおわかりいただけると思います。

ib_udf.c(238)

```
double EXPORT IB_UDF_rand()
{
    srand((unsigned) time(NULL));
    return ((float) rand() / (float) RAND_MAX);
}
```

見てわかるとおり、IB_UDF_rand() 関数は引数をとらず、戻り値が double 型となっています。

1 行目で、srand() 関数によって乱数系列を初期化しています。RAND_MAX は stdlib.h で定義されている定数値で、rand() 関数が 0-RAND_MAX の範囲の乱数を返すため、RAND_MAX で割ることで 0-1 の範囲の乱数を返すということになります。

問題は、IB_UDF_rand() 関数の呼出を行う度に、srand() 関数が呼ばれる事にあります。あるいは、Windows のタイマの精度が低いためとも言えるのですが、連続した関数呼び出し中で time() 関数が同じ値を返してしまい、結果として rand() 関数が同じ値を返す事になってしまいます。

そこで、傾向と対策 3 ではテーブルにランダムなサンプルデータを作成するために、以下のような UDF を作成して、乱数を取得する方法を示しました。(呼出規約が cdecl だったので、これを stdcall に変更してあります。また、uses 節から Classes を省略しました。こうすると 100kb くらいサイズが小さくなります。)

TomekoUDF.pas

```
library TomekoUDF;

uses
    SysUtils;

function Tom_Rand: Double; stdcall;
begin
    //Randomize; //ここで Randomize しては駄目
    Result := Random;
end;

function Tom_Randmize: Integer; stdcall;
begin
    Randomize;
    Result := 1;
end;

exports
    Tom_Rand,
    Tom_Randmize;
begin
end.
```

こちらも一見して非常に単純です。パラメータを受け取らず、戻り値が数値型の UDF は上記のような書き方をする事になります。呼出規約は、InterBase6 以降では stdcall を使うように指定されています。

これをコンパイルするためには、以下のようにします。

```
>dcc32 TomekoUDF.pas
Borland Delphi Version 14.0
Copyright (c) 1983, 2002 Borland Software Corporation
tomkoudf.pas(29)
30 lines, 0.24 seconds, 28080 bytes code, 3157 bytes data.
```

作成された UDF の登録用スクリプトは以下のようになります。

```
/* function Tom_Rand: double;
*/
```

```
DECLARE EXTERNAL FUNCTION TOM_RAND
```

```
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'Tom_Rand' MODULE_NAME 'TOMEKOUDF';
```

```
/* function Tom_Randmi ze: integer;
*/
```

```
DECLARE EXTERNAL FUNCTION TOM_RANDOMIZE
RETURNS INTEGER BY VALUE
ENTRY_POINT 'Tom_Randmi ze' MODULE_NAME 'TOMEKOUDF';
```

従って、UDFを単純化してみると次のように書けばよいということになります。

最も単純な UDF :

```
function hoge hoge: integer; stdcall;
begin
  result := 1;
end;
```

同様に、円周率を返す関数は以下のように書くことが出来ます。

ソースコード :

```
library TomekoUDF2;
```

```
uses
  SysUtils;
```

```
{$R *.res}
```

```
function Tom_Pi: double; stdcall;
begin
  Result := Pi;
end;
```

```
exports
  Tom_Pi;
```

```
begin
end.
```

登録用スクリプト :

```
/* External Function declarations */
```

```
DECLARE EXTERNAL FUNCTION TOM_PI
RETURNS DOUBLE PRECISION BY VALUE
```

```
ENTRY_POINT 'Tom_Pi' MODULE_NAME 'TomekoUDF2';
```

使用例 :

```
SQL> SELECT TOM_PI () FROM RDBSDATABASE;
          TOM_PI
```

```
=====
          3.141592653589793
```

引数と戻り値

次に、UDFに引数を渡すための方法を検討します。まず、Firebirdでは、UDFに10個までの引数を渡すことが出来ます。ただし、配列要素は引数として渡すことは出来ません。BLOB型については、後ほど検討します。

プログラミング言語での引数のデータ型とFirebirdのデータ型とは一致しなくても構いませんが、Firebirdが取り扱える形式でなくてはなりません。たとえば、先ほどのTomPi()関数は、DelphiではDouble型ですが、Firebirdでは自動的にDOUBLE PRECISION型に変換されます。

UDFの戻り値については、デフォルトでは参照渡しで返されますが、数値型については値渡しを指定することも可能です。その場合は、DECLARE EXTERNAL FUNCTION文でBY VALUEキーワードを指定します。また、戻り値の指定でRETURNS PARAMETER nを利用することで、Firebirdの管理するメモリ領域を利用して戻り値を返すことが可能です。この点についても、後ほど検討します。

引数を渡す方法(数値型)

さて、UDFに引数を渡すコードを見てみましょう。同じくIB_UDFのソースコードからです。

```
ib_udf.c(48)
double EXPORT IB_UDF_abs( double *a)
{
    return (*a < 0.0) ? -*a : *a;
}
```

これも大変単純な関数ですが、IB_UDF_abs()関数は、引数の絶対値を返します。引数はdouble型のポインタaとして宣言されています。三項演算子を使用して、引数が0.0より小さければ-aを、そうでなければaをそのまま返します。

同じUDFをDelphiで作成すると以下ようになります。

```
library TomekoUDF3;

uses
    SysUtils;

{$R *.res}

function TomAbs(var v: double): double; stdcall;
begin
    Result := Abs(v);
end;

exports
    TomAbs;

begin
end.
```

登録用スクリプト:

```
/* External Function declarations */

DECLARE EXTERNAL FUNCTION TOM_Abs
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'TomAbs' MODULE_NAME 'TomekoUDF3';
```

使用例:

```
SQL> SELECT TomAbs(-3) FROM RDB$DATABASE;
          TOM_ABS
=====
          3.000000000000000
```

次に、Excelでおなじみのセンチメートルからポイントへ、また、ポイントからセンチメートルへ、単位を変換するUDFを作成してみます。Convert()関数は、第一引数がconst指定されているため、そのまま引数vを渡すことが出来ません。そこで、double型の変数dを宣言して、一度代入してから渡しています。

```
library TomekoUDF4;
```

```
uses
    SysUtils
    stdConv,
    ConvUtils;

{$R *.res}
```

```
function Tom_CmToPts(var v: double): double; stdcall;
var
  d: double;
begin
  d := v;
  Result := Convert(d, duCentimeters, duPoints);
end;
```

```
function Tom_PtsToCm(var v: double): double; stdcall;
var
  d: double;
begin
  d := v;
  Result := Convert(d, duPoints, duCentimeters);
end;
```

```
exports
  Tom_CmToPts,
  Tom_PtsToCm
```

```
begin
end. library TomnekoUDF3;
```

登録用スクリプト:

```
/* External Function declarations */
```

```
DECLARE EXTERNAL FUNCTION TOM_CmToPts
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'Tom_CmToPts' MODULE_NAME 'TomnekoUDF4';
```

```
DECLARE EXTERNAL FUNCTION TOM_PtsToCm
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'Tom_PtsToCm' MODULE_NAME 'TomnekoUDF4';
```

使用例:

```
SQL> SELECT Tom_CmToPts(2) FROM RDBSDATABASE;
          TOM_CMFOPTS
```

```
=====
          56.90551238007875
```

```
SQL> SELECT Tom_PtsToCm(56.90551238007875) FROM RDBSDATABASE;
          TOM_PTSTOCM
```

```
=====
          2.000000000000000
```

引数を渡す方法(文字型)

次に、UDFに文字型のデータを渡すコードを見てみましょう。同じくIB_UDFのソースコードからです。

```
ib_udf.c(68)
int EXPORT IB_UDF_ascii_val( char *a)
{
    return ((int) (*a));
}
```

IB_UDF_ascii_val()関数は、引数で指定されたASCII文字の文字コードを返す関数です。char型のポインタaで参照される文字をint型にキャストして返しているだけです。シンプルですね。

同じ事をDelphiで書くと以下ようになります。

```
library TomekoUDF5;

uses
    SysUtils;

{$R *.res}

function Tom_ascii_val(var a:char):Integer;stdcall;
begin
    Result := Integer(a);
end;

exports
    Tom_ascii_val;

begin
end.
```

登録用スクリプト:

```
/* External Function declarations */

DECLARE EXTERNAL FUNCTION TOM_ascii_val
CHAR(1) CHARACTER SET ASCII
RETURNS INTEGER BY VALUE
ENTRY_POINT 'Tom_ascii_val' MODULE_NAME 'TomekoUDF5';
```

使用例:

```
SQL> SELECT Tom_ascii_val('a') FROM RDBSDATABASE;
TOM_ASCII_VAL
```

```
=====
```

文字型のデータを戻り値として返す(1)

さて、`ascii_val()` 関数のように、文字列を引数として受け取るような場合は問題とならないのですが、文字列を戻り値として返すために、関数の内部で文字列処理のメモリ領域を用意するような場合は注意が必要です。

ご存じのように **Firebird** のスーパーサーバ版はマルチスレッド化されているため、複数の接続に対して 1 プロセスの **Firebird** サーバーが複数のスレッドを生成して処理を行っています。

この時、各スレッドはメモリ空間を共有しているため、UDF の呼び出しにおいて同じ関数が別のスレッドから同時に呼ばれてしまった場合、静的変数の内容が上書きされてしまうなどの再入問題が発生する可能性があります。ところが、動的にメモリ領域を確保するとすると、誰がそれを解放するのかという問題が生じます。UDF 内部で確保されたメモリ領域を使って、戻り値を **Firebird** へ渡す場合、**Firebird** がそのメモリ領域を開放しなくてはならないからです。

この問題の解決方法として、**Firebird** には、`ib_util.dll` (Linux では `ib_util.so`) というライブラリが付属しています。これは、UDF 内で動的にメモリを確保するために使用されます。そうすることで、呼び出し側である **Firebird** がそのメモリを解放することが可能となります。

具体的には、Delphi では **Firebird** のインストールディレクトリ配下の `include` ディレクトリにある、`ib_util.pas` を `uses` して、`IB_UTIL_malloc()` 関数を利用することになります。C 言語では、`ib_util.h` をインクルードすることになっているのですが、Windows 版の **Firebird** 1.0.3 では、インストールされないようです。**Firebird** 1.5.4290 では両方とも `include` ディレクトリにインストールされています。

※注意：

Delphi 用の Unit である、`ib_util.pas` は、実はそのままでは使えません。ソースコード中の `IB_UTIL_malloc()` 関数の定義を全て小文字にする必要があります。この問題は **Firebird** 1.5 では修正済です。

```
function IB_UTIL_malloc(l: integer): pointer; cdecl;
```

```
function ib_util_malloc(l: integer): pointer; cdecl;
```

さて、では `ib_udf` のコードをみてみましょう。

```
ib_udf.c(48)
char *EXPORT IB_UDF_ascii_char( int *a)
{
    char *b;
    b = (char *) ib_util_malloc(2);
    *b = (char) (*a);
    /* let us not forget to NULL terminate */
    b[1] = '\0';
    return (b);
}
```

`IB_UDF_ascii_char()` 関数は、先ほどの `IB_UDF_ascii_val()` 関数とは逆に、ASCII コードから ASCII 文字を返す関数です。

`int` 型の値を引数として受け取り、`char` 型のポインタを宣言して、`ib_util_malloc()` 関数で動的にメモリを確保しています。

そして、`b` に対して、`a` の値を `char` 型にキャストしてマルチターミナートを行い、`b` を返しています。`ib_util_malloc()` 関数で確保した領域を、解放していないことに注意して下さい。

登録用スクリプト：

```
DECLARE EXTERNAL FUNCTION ascii_char
INTEGER
RETURNS CSTRING(1) FREE_IT
ENTRY_POINT 'IB_UDF_ascii_char' MODULE_NAME 'ib_udf';
```

使用例：

```
SQL> SELECT ascii_char(97) FROM RDBSDATABASE;
ASCII_CHAR
=====
a
```

登録用スクリプトの `REURNS` 句の最後に `FREE IT` キーワードが指定されています。この `FREE IT` 指定によって、**Firebird** は UDF の呼び出しが終了した時点で、不要となったメモリ領域を `ib_util_malloc()` と同じメモリマネージャーを利用して解放します。

では、Delphi で、同じ事をやってみましょう。

```
library TomnekoUDF6;
```

```
uses
```

```
    SysUtils,
    ib_util;
```

```
{SR *.res}
```

```
function Tom_ascii_char(var a:Integer):PChar;stdcall;
begin
```

```
result := ib_util_malloc(2);
result^ := Char(a);
(result + 1)^ := #0;
end;
```

```
exports
  Tom_ascii_char;
```

```
begin
end.
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION Tom_ascii_char
INTEGER
RETURNS CSTRING(1) CHARACTER SET ASCII FREE_IT
ENTRY_POINT 'Tom_ascii_char' MODULE_NAME 'TomnekoUDF6';
```

使用例 :

```
SQL> SELECT TOM_ASCII_CHAR(97) FROM RDBSDATABASE;
```

```
TOM_ASCII_CHAR
```

```
=====
```

```
a
```

文字型のデータを戻り値として返す(2)

さて、`ib_util_malloc()`関数と `FREE IT` キーワードを利用する方法は、しかし余りスマートとは言えません。そこで、もう一つの方法があります。

`DECLARE EXTERNAL FUNCTION` 文の `RETURNS` 句で戻り値のデータ型を指定する代わりに、`PARAMETER` 句を指定することで、引数として渡すデータの中から戻り値を指定することができます。具体的には、以下のようにします。

```
library TomnekoUDF8;

uses
  SysUtils;

{SR *.res}

procedure Tom_ascii_char2(var a: Integer; b: PChar); stdcall;
begin
  b^ := Char(a);
  (b+1)^ := #0;
end;

exports
  Tom_ascii_char2;

begin
end.
```

登録用スクリプト :

```
DECLARE EXTERNAL FUNCTION Tom_ascii_char2
INTEGER,
CSTRING(1) CHARACTER SET ASCII
RETURNS PARAMETER 2
ENTRY_POINT 'Tom_ascii_char2' MODULE_NAME 'TomnekoUDF8';
```

使用例 :

```
SQL> select tom_ascii_char2(97) from rdb$database;
TOM_ASCII_CHAR2
=====
a
```

この場合、戻り値のために必要なメモリ領域は、`Firebird` が管理しているので、`FREE IT` キーワードも `ib_util_malloc()` 関数も必要ありません。

日付時刻型の扱い

Firebird の内部における日付時刻型は、そのままでは C や Delphi で利用することが出来ません。そのため、以下の API を利用して Firebird から渡された日付時刻を変換し、また、戻り値として渡さなくてはなりません。

`isc_decode_sql_date()` :

Firebird の内部日付形式を C の日付構造体に変換する

`isc_encode_sql_date()` :

C の日付構造体を Firebird の内部日付形式に変換する

`isc_decode_sql_time()` :

Firebird 内部時刻形式を C の時刻構造体に変換する

`isc_encode_sql_time()` :

C の時刻構造体を Firebird の内部時刻形式に変換する

`isc_decode_timestamp()` :

Firebird の内部タイムスタンプ形式を C のタイムスタンプ構造体に変換する
(従来の `isc_decode_date()` 呼び出し)

`isc_encode_timestamp()` :

C のタイムスタンプ構造体を Firebird の内部タイムスタンプ形式に変換する
(従来の `isc_encode_date()` 呼び出し)

`ib_udf` ライブラリには日付時刻型の関数は含まれていないので、`fbudf` ライブラリのコードを見てみることにしましょう。

`fbudf.cpp` (462)

```
FBUDF_API ISC_TIMESTAMP* addYear(ISC_TIMESTAMP* v, int& nyears)
{
    tm times;
    isc_decode_timestamp(v, &times);
    times.tm_year += nyears;
    isc_encode_timestamp(&times, v);
    return v;
}
```

まず、C の時刻型である `tm` 型の `times` を宣言し、`ISC_TIMESTAMP` 型で渡されてきた `v` を `isc_decode_timestamp()` 関数で C の時刻型に変換しています。そして、`tm` 型のメンバである `tm_year` に `int` 型で渡された `nyear` を加算しています。その `times` を `isc_encode_timestamp()` 関数を使って再び Firebird の `Timestamp` 型へ変換して戻しています。

では、Delphi で同じ事をしてみます。

library TomnekoUDF9;

uses

SysUtils, Windows;

type

TM = record

```
    tm_sec : integer; // Seconds
    tm_min : integer; // Minutes
    tm_hour : integer; // Hour (0--23)
    tm_mday : integer; // Day of month (1--31)
    tm_mon : integer; // Mnth (0--11)
    tm_year : integer; // Year (calendar year minus 1900)
    tm_wday : integer; // Weekday (0--6) Sunday = 0)
    tm_yday : integer; // Day of year (0--365)
    tm_isdst : integer; // 0 if daylight savings time is not in effect)
```

end;

PTM = ^TM

Long = Longint;

ULong = Longword;

ISC_TIMESTAMP = record

```
    timestamp_date : Long;
    timestamp_time : ULong;
```

end;

PISC_TIMESTAMP = ^ISC_TIMESTAMP;

procedure isc_encode_timestamp

(tm_date: PTM; ib_date: PISC_TIMESTAMP);

```

stdcall; external 'GDS32.DLL';

procedure isc_decode_timestamp
(ib_date: PISC_TIMESTAMP; tm_date: PTM);
stdcall; external 'GDS32.DLL';

procedure isc_decode_sql_date
(var ib_date: Long; tm_date: PTM);
stdcall; external 'GDS32.DLL';

procedure isc_encode_sql_date
(tm_date: PTM; var ib_date: Long);
stdcall; external 'GDS32.DLL';

procedure isc_decode_sql_time
(var ib_date: ULong; tm_date: PTM);
stdcall; external 'GDS32.DLL';

procedure isc_encode_sql_time
(tm_date: PTM; var ib_date: ULong);
stdcall; external 'GDS32.DLL';

{SR *.res}

function Tom_addYear(var v: ISC_TIMESTAMP; var nyears: Integer): PISC_TIMESTAMP; stdcall;
var
  times: tm;
begin
  isc_decode_timestamp(@v, @times);
  times.tm_year := times.tm_year + nyears;
  isc_encode_timestamp(@times, @v);
  result := @v;
end;

exports
  Tom_addYear;

begin
end.

```

登録用スクリプト :

```

DECLARE EXTERNAL FUNCTION Tom_addYear
TIMESTAMP, INTEGER
RETURNS TIMESTAMP
ENTRY_POINT 'Tom_addYear'
MODULE_NAME 'TomekoUDF9';

```

使用例 :

```

SQL> CREATE TABLE T_DATE (TM DATE);
SQL> INSERT INTO T_DATE VALUES ('2004/1/1');
SQL> SELECT TOM_ADDYEAR(TM 2) FROM T_DATE;
          TOM_ADDYEAR
=====
2006-01-01 00:00:00.0000

```

この場合、Firebirdから渡されたvをそのままresultとしているところがみそで、戻り値用にtm型のメモリを確保しないようにしているわけです。

UDF で BLOB を扱う

UDF で BLOB を扱う場合、いくつかの注意が必要です。BLOB が UDF に渡される、あるいは返される場合、それは以下に示す BLOB 制御構造体へのポインタとなります。BLOB の実データに対しては、この BLOB 制御構造体を通してアクセスすることになります。

BLOB 制御構造体 : C 言語の定義 :

```
fun.e(91)
static SSHORT
    blob_get_segment (BLB, UCHAR *, USHORT, USHORT *);
static void
    blob_put_segment (BLB, UCHAR *, USHORT);

val.h(120)
typedef struct blob {
    short (*blob_get_segment) ();
    void *blob_handle;
    long blob_number_segments;
    long blob_max_segment;
    long blob_total_length;
    void (*blob_put_segment) ();
} *BLOB;
```

BLOB 制御構造体 : Delphi の定義 :

```
IBExternal.pas(38)
type
    Long          = LongInt; { 32 bit signed }
    Short         = SmallInt; { 16 bit signed }
    PInt          = ^Int;

    TISC_BlobGetSegment =
        function (BlobHandle: PInt;
                 Buffer: PChar;
                 BufferSize: Long;
                 var ResultLength: Long): Short; cdecl;
    TISC_BlobPutSegment =
        procedure (BlobHandle: PInt;
                  Buffer: PChar;
                  BufferLength: Short); cdecl;

    TBlob = record
        GetSegment      : TISC_BlobGetSegment;
        BlobHandle      : PInt;
        SegmentCount    : Long;
        MaxSegmentLength : Long;
        TotalSize       : Long;
        PutSegment      : TISC_BlobPutSegment;
    end;
    PBlob = ^TBlob;
```

※Delphi での定義は、IPL に基づいて公開されている IBX のソースコードから必要な部分のみ引用しています。

BLOB 構造体の各フィールドはそれぞれ以下のような機能を持っています。

blob_get_segment :

BLOB 制御構造体の最初のフィールドである **blob_get_segment** は、読み取り用の関数へのポインタとなっています。UDF から BLOB を読み取る場合は、この関数を呼び出して、BLOB からセグメントを読み取ります。

blob_get_segment の引数は 4 つあり、BLOB ハンドル、読み取りバッファを示す Char 型のポインタ (このバッファに BLOB のセグメントが格納されます)、読み取りバッファのサイズ、実際に読み取ったバイト数を格納する変数のアドレスとなります。UDF による BLOB データの読み取りが行われない場合、**blob_get_segment** を NULL に設定します。

blob_handle :

BLOB 制御構造体の 2 番目のフィールドは必須項目です。**blob_handle** は、UDF に渡される BLOB の特定に使用され、また、UDF から返される BLOB の特定に使用されます。**isc_blob_handle** 型は、汎用ポインタです。

number_segments :

UDF に渡された BLOB データのセグメントの総数が **number_segments** に格納されています。UDF による BLOB データの読み取りが行われなかったときには、**number_segments** を NULL に設定します。

max_seglen :

UDF に渡された BLOB データのセグメントの最大長がバイト数で **max_seglen** に格納されています。UDF による BLOB データの読み取りが行われなかった場合、**max_seglen** を NULL に設定します。

total_size :

UDF に渡された BLOB データの全体のサイズがバイト数で **total_size** に格納されています。UDF による BLOB データの読み取りが行われなかった場合、**max_seglen** を NULL に設定します。

blob_put_segment :

BLOB 制御構造体の最後の項目である **blob_put_segment** は、書き込み用の関数へのポインタとなっています。UDF から **BLOB** を返す場合には、この関数を使用して **BLOB** にセグメントを書き込みます。

blob_put_segment の引数は 3 つあり、**BLOB** ハンドル、書込バッファを示す **Char** 型のポインタ（このバッファが **BLOB** のセグメントに書き込まれます）、書き込まれるデータのサイズ（バイト数）となっています。

BLOB UDFのサンプル(1)

それでは、BLOBを扱うUDFの例を見ていきましょう。まずは、Ian Newby氏作のBLOB UDFsから、一番簡単なサンプルを示します。

```
wmudflib.c(35)
long EXPORT fn_blob_length (ARG(BLOB, sourceBlob))
ARGLIST(BLOB sourceBlob) {
    if (!sourceBlob->blob_handle) {
        return 0L;
    }
    return (sourceBlob->blob_total_length);
}
```

blob_handleがNULLであれば、0を返して抜ける。そうでない場合は、sourceBlobのblob_total_lengthを返すというものです。

Delphiで同じ事をやってみます。

```
library TomnekoUDF12;

uses
    SysUtils,
    IExternalS;

{$R *.res}

function Tom_Blob_Length(SourceBlob: PBlob): Integer; stdcall;
begin
    Result := 0;
    if (SourceBlob^.BlobHandle^ = 0) then
        Exit
    else
        Result := SourceBlob^.TotalSize;
end;

exports
    Tom_Blob_Length;

begin
end.
```

登録用スクリプト:

```
DECLARE EXTERNAL FUNCTION Tom_Blob_Length
BLOB
RETURNS INTEGER BY VALUE
ENTRY_POINT 'Tom_Blob_Length'
MODULE_NAME 'TomnekoUDF12';
```

使用例:

```
SQL> SELECT TOM_BLOB_LENGTH(BLOB1) FROM T_BLOB;
```

TOM_BLOB_LENGTH

=====

```
89334
89334
89334
89334
89334
```

BLOB UDF のサンプル(2)

次に、同じく Ian Newby 氏作の BLOB UDFs から、BLOB から文字列への変換を行う UDF と、逆に文字列から BLOB を返す UDF を見てみます。

```
wmdflib.c(100)
char* EXPORT fn_blob_string(ARG(BLOB, sourceBlob), ARG(char*, sResult))
ARGLIST(BLOB sourceBlob)
ARGLIST(char *sResult) {
    long start;
    long end;
    start = 1;
    end = fn_blob_length(sourceBlob);
    fn_blob_substr(sourceBlob, &start, &end, sResult);
    return sResult;
}
```

fn_blob_string()関数は、BLOB を文字列として返します。実際に BLOB に格納されているデータが文字列かどうかの判断はしていないので、注意が必要です。

内容的には、先ほど示した fn_blob_length() 関数を使って blob の長さを得てから、fn_blob_substr() 関数を利用して最初から最後までまでのデータを取り出しています。fn_blob_substr() 関数は以下のような内容です。

```
wmdflib.c(47)
char* EXPORT fn_blob_substr(ARG(BLOB, sourceBlob), ARG(long*, startPos), ARG(long*, endPos), ARG(char*, sResult))
ARGLIST(BLOB sourceBlob)
ARGLIST(long *startPos)
ARGLIST(long *endPos)
ARGLIST(char *sResult) {
    char *pbuffer, *poffset, *pResultOffset;
    long i = 0;
    long curr_bytecount = 0;
    long startChar, endChar;
    long length, actual_length;
    *sResult = 0;
    if (!sourceBlob->blob_handle) {
        return sResult;
    }
    length = sourceBlob->blob_max_segment;
    if (*startPos > *endPos || *startPos < 1L || *endPos < 1L) {
        return sResult;
    }
    if (sourceBlob->blob_total_length < (long)*startPos) {
        return sResult;
    }
    startChar = *startPos;
    if (sourceBlob->blob_total_length < (long)*endPos) {
        endChar = sourceBlob->blob_total_length;
    } else {
        endChar = *endPos;
    }
    pbuffer = (char *) malloc (length + 1L);
    pResultOffset = sResult;
    while ((*sourceBlob->blob_get_segment) (sourceBlob->blob_handle, pbuffer, length, &actual_length)) {
        // pbuffer [actual_length] = 0;
        poffset = pbuffer;
        while (*poffset && (curr_bytecount < endChar)) {
            curr_bytecount++;
            if (curr_bytecount >= startChar) {
                *pResultOffset++ = *poffset;
            }
            poffset++;
        }
        if (curr_bytecount >= endChar) {
            *pResultOffset = 0;
            break;
        }
    }
    free (pbuffer);
    return sResult;
}
```

fn_blob_substr()関数では、引数のチェックを行った後、sourceBlobで渡されたBLOB制御構造体のblob_get_segmentを利用して、BLOBのデータを取り出し、sResultで渡される戻り値へコピーを行っています。

```
wmudflib.c(153)
BLOB_EXPORT fn_string_blob(ARG(char*, sourceString), ARG(BLOB, sResult))
ARGLIST(char *sourceString)
ARGLIST(BLOB sResult) {
    (*sResult->blob_put_segment) (sResult->blob_handle, sourceString, strlen(sourceString));
    return sResult;
}
```

fn_string_blob()関数は、単純に、BLOB制御構造体のblob_put_segment()関数を利用して、BLOBにデータを書き込んでFirebirdへ返しているだけです。

```
library TomekoUDF12;
```

```
uses
    SysUtils,
    IBEexternals;
```

```
{SR *.res}
```

```
function Tom_Blob_Length(SourceBlob: PBlob): Integer; stdcall;
begin
    Result := 0;
    if (SourceBlob^.BlobHandle = 0) then
        Exit
    else
        Result := SourceBlob^.TotalSize;
end;
```

```
Procedure Tom_Blob_substr(SourceBlob: PBlob; var startPos, endPos: Integer; sResult: PChar); stdcall;
var
    pBuffer, pResultOffset, pOffset: PChar;
    curr_bytecount, startChar, endChar, length, actual_length: Integer;
begin
    curr_bytecount := 0;
    sResult^ := #0;
    if (sourceBlob^.BlobHandle = 0) then
        begin
            Exit;
        end;
    length := sourceBlob^.MaxSegmentLength;

    if ((startPos > endPos) or (startPos < 1) or (endPos < 1)) then
        begin
            exit;
        end;
    if (sourceBlob^.TotalSize < startPos) then
        begin
            exit;
        end;
    startChar := startPos;

    if (sourceBlob^.TotalSize < endPos) then
        begin
            endChar := sourceBlob^.TotalSize;
        end else
        begin
            endChar := endPos;
        end;

    GetMem(pBuffer, length + 1);

    pResultOffset := sResult;
    while (sourceBlob^.GetSegment(sourceBlob^.BlobHandle, pBuffer, length, actual_length) <> 0) do
        begin
            pOffset := pBuffer;
            while ((pOffset <> #0) and (curr_bytecount < endChar)) do
                begin
```

```

    Inc(curr_bytecount);
    if (curr_bytecount >= startChar ) then
    begin
        pResultOffset^ := pOffset^;
        inc(pResultOffset);
    end;
    Inc(pOffset);
    if (curr_bytecount = actual_length) then Break;
end;

    if (curr_bytecount >= endChar) then
    begin
        pResultOffset^ := #0;
        break;
    end;
end;
freeMem(pbuffer);
exit;
end;

Procedure Tom_Blob_string(sourceBlob:PBlob; sResult:PChar);stdcall;
var
    startPos, endPos:Integer;
begin
    startPos := 1;
    endPos := Tom_blob_length(sourceBlob);
    Tom_blob_substr(sourceBlob, startPos, endPos, sResult);
end;

Procedure Tom_string_Blob(sourceString:PChar; sResult:PBlob);stdcall;
begin
    sResult^.PutSegment(sResult^.BlobHandle, sourceString, strlen(sourceString));
end;

exports
    Tom_Blob_Length,
    Tom_Blob_substr,
    Tom_Blob_string,
    Tom_string_Blob;

begin
end.

```

登録用スクリプト :

```

DECLARE EXTERNAL FUNCTION Tom_BLOB_SUBSTR
    BLOB,
    INTEGER,
    INTEGER,
    CSTRING(256) CHARACTER SET ASCII
    RETURNS PARAMETER 4
    ENTRY_POINT 'Tom_Blob_substr'
    MODULE_NAME 'TomekoUDF12';

DECLARE EXTERNAL FUNCTION Tom_BLOB_TO_STRING
    BLOB,
    CSTRING(256) CHARACTER SET ASCII
    RETURNS PARAMETER 2
    ENTRY_POINT 'Tom_Blob_string'
    MODULE_NAME 'TomekoUDF12';

DECLARE EXTERNAL FUNCTION Tom_STRING_TO_BLOB
    CSTRING(32000) CHARACTER SET ASCII,
    BLOB
    RETURNS PARAMETER 2
    ENTRY_POINT 'Tom_string_Blob'
    MODULE_NAME 'TomekoUDF12';

```

使用例 :

```

SQL> CREATE TABLE T_BLOB (BLOB1 BLOB SUB_TYPE 1);
SQL> INSERT INTO T_BLOB

```

```
VALUES (TOM_STRING_TO_BLOB('TOMNEKO'));
SQL> SELECT TOM_BLOB_TO_STRING(BLOB1) FROM T_BLOB;
TOM_BLOB_TO_STRING
=====
TOMNEKO
```

Blob フィルタ

BLOB フィルタはあるサブタイプの BLOB データを他のサブタイプに変換するものです。Firebird には、サブタイプ 0(未定義)をサブタイプ 1(TEXT)に変換・逆変換する BLOB フィルタが組み込まれています。

例としては、小文字のテキストを大文字に変換するとか、Bitmap ファイルを Jpeg に変換する等が挙げられます。

Blob フィルタのスケルトン

IBPhoenix 社のサイトからダウンロードすることができる「InterBase 6.0 Beta Documentation」のうち、InterBase 6.0 API Guide に BLOB フィルタに関する解説が記載されています。P. 138 に示されている BLOB フィルタのスケルトンを以下に示します。

```
#include <ibase.h>
#define SUCCESS 0
#define FAILURE 1
ISC_STATUS jpeg_filter(short action, isc_blob_ctl control)
{
    ISC_STATUS status = SUCCESS;
    switch (action)
    {
        case isc_blob_filter_open:
            . . .
            break;
        case isc_blob_filter_get_segment:
            . . .
            break;
        case isc_blob_filter_create:
            . . .
            break;
        case isc_blob_filter_put_segment:
            . . .
            break;
        case isc_blob_filter_close:
            . . .
            break;
        case isc_blob_filter_alloc:
            . . .
            break;
        case isc_blob_filter_free:
            . . .
            break;
        case isc_blob_filter_seek:
            . . .
            break;
        default:
            . . .
            break;
    }
}
```

BLOB フィルタには filter_function() が必要です。これは、必ず以下の形式に従わなくてはなりません。

```
ISC_STATUS filter_function_name(short action, isc_blob_ctl control);
```

そして、filter_function には、action のそれぞれに対する処理を記述する事になります。action については、ibase.h で定義されています。

```
#define isc_blob_filter_open 0
#define isc_blob_filter_get_segment 1
#define isc_blob_filter_close 2
#define isc_blob_filter_create 3
#define isc_blob_filter_put_segment 4
#define isc_blob_filter_alloc 5
#define isc_blob_filter_free 6
#define isc_blob_filter_seek 7
```

それぞれの意味については以下の通りです。

```
isc_blob_filter_open :
    アプリケーションが isc_blob_open2() API を呼び出した時に起動します
```

```
isc_blob_filter_get_segment :
    アプリケーションが isc_get_segment() API を呼び出した時に起動します
```

isc_blob_filter_close :

アプリケーションが `isc_close_blob()` API を呼び出した時に起動します

isc_blob_filter_create :

アプリケーションが `isc_create_blob2()` API を呼び出した時に起動します

isc_blob_filter_put_segment :

アプリケーションが `isc_put_segment` を呼び出した時に起動します

isc_blob_filter_alloc :

Firebird がフィルタ処理を開始する際に起動します。アプリケーションの動作とは関係ありません

isc_blob_filter_free :

Firebird がフィルタ処理を終了する際に起動します。アプリケーションの動作とは関係ありません

isc_blob_filter_seek :

内部でのフィルタ処理に予約されています。外部のフィルタでは使用されません

また、`filter function()` の第二引数である `isc_blob_ctl` 構造体は同じく `ibase.h` で以下のように定義されています。

```
typedef struct isc_blob_ctl {
    ISC_STATUS (*ctl_source)();
    /* Internal InterBase Blob access routine. */
    struct isc_blob_ctl *ctl_source_handle;
    /* Instance of isc_blob_ctl to pass to
    internal InterBase Blob access routine. */
    short ctl_to_sub_type; /* Target subtype. */
    short ctl_from_sub_type; /* Source subtype. */
    unsigned short ctl_buffer_length; /* Length of ctl_buffer. */
    unsigned short ctl_segment_length; /* Length of current segment. */
    unsigned short ctl_bpb_length; /* Blob parameter buffer length. */
    char *ctl_bpb; /* Pointer to Blob parameter buffer. */
    unsigned char *ctl_buffer; /* Pointer to segment buffer. */
    ISC_LONG ctl_max_segment; /* Length of longest Blob segment. */
    ISC_LONG ctl_number_segments; /* Total number of segments. */
    ISC_LONG ctl_total_length; /* Total length of Blob. */
    ISC_STATUS *ctl_status; /* Pointer to status vector. */
    long ctl_data[8]; /* Application-specific data. */
} *ISC_Blob_CTL;
```

`isc_blob_ctl` 構造体のそれぞれのフィールドは、Firebird からフィルタ関数へ値を渡す (IN) ために利用されるものと、フィルタ関数から Firebird へ値を渡す (OUT) ために利用されるものが混在しています。それぞれの詳細は以下の通りです。

(*ctl_source)() :

(IN) : Firebird の内部 Blob アクセスルーチンへのポインタ

***ctl_source_handle Pointer :**

(IN) : Firebird の内部 Blob アクセスに渡される `isc_blob_ctl` のインスタンスへのポインタ

ctl_to_sub_type :

(IN) : 変換先のサブタイプ。複数のサブタイプに対応した BLOB フィルタはこの情報と、`ctl_from_sub_type` フィールドの値を元に動作を決定する。

ctl_from_sub_type :

(IN) : 変換元のサブタイプ。複数のサブタイプに対応した BLOB フィルタはこの情報と、`ctl_to_sub_type` フィールドの値を元に動作を決定する。

ctl_buffer_length :

(IN) : `isc_blob_filter_put_segment` からの呼出の場合は、IN フィールドで `ctl_buffer` に格納されたセグメントデータ長が格納されてくる

(OUT) : `isc_blob_filter_get_segment` からの呼出の場合は、OUT フィールドで `ctl_buffer` が示すバッファのサイズを格納します。バッファには取り出したデータを格納しておきます。

ctl_segment_length :

(OUT) : 現在のセグメント長を格納します。`isc_blob_filter_get_segment` からの呼出では、取り出したセグメント長を格納します

(セグメントの一部が取り出された場合には、`ctl_buffer_length` で示されるバッファ長は実際のセグメント長より小さくなります)

`isc_blob_filter_put_segment` からの呼出では使用されません。

ctl_bpb_length :

(IN) : BLOB パラメータバッファ (BPB) 長が格納されます

***ctl_bpb :**

(IN) : BLOB パラメータバッファ (BPB) へのポインタが格納されます

***ctl_buffer :**

(IN) : セグメントバッファへのポインタ。isc_blob_filter_put_segment からの呼出では IN フィールドになり、セグメントデータの格納されたバッファへのポインタが格納されます。

(OUT) : isc_blob_filter_get_segment からの呼出では OUT フィールドになり、フィルタ関数によってアプリケーションに返すセグメントデータを格納したバッファへのポインタとなります。

ctl_max_segment :

(OUT) : BLOB 中で最も長いセグメントの長さをバイト単位で格納します。初期値は 0 で、フィルタ関数が値を設定します。このフィールドは情報提供のためにのみ使用されています。

ctl_number_segments :

(OUT) : BLOB 中のセグメントの総数です。初期値は 0 で、フィルタ関数が値を設定します。このフィールドは情報提供のためにのみ使用されています。

ctl_total_length :

(OUT) : BLOB 全体の長さをバイト数で格納します。初期値は 0 で、フィルタ関数が値を設定します。このフィールドは情報提供のためにのみ使用されています。

***ctl_status :**

(OUT) : エラー情報等を格納するステータスペクターへのポインタ。

ctl_data[8] :

(IN/OUT) : アプリケーションが独自に使用可能な 8 要素の配列。このフィールドにはメモリへのポインタや

isc_blob_filter_open ハンドラで生成したファイルへのハンドル等を格納するために使用できます。格納した値は、次回フィルタ関数が呼ばれる際に Firebird から戻されます。

Firebird のオブジェクト命名規則

Firebird でテーブル、フィールド、トリガー等の名前を付けるさいには、以下の命名規則に従わなくてはなりません。

- スペースを含めることはできません。
 - 大文字と小文字は区別されません。
 - **Firebird** の予約語と同じであってははいけません。
- ただし、**Dialect3** のデータベースで、オブジェクトの名前を二重引用符で囲った場合はこの限りではありません。その場合、次のようなことが可能となります。
このような名前を区切付識別子 (**Delimited Identifier**) と呼びます。
- a) **Firebird** の予約語を使用できます。
 - b) 名前にスペースを含めることができます。ただし、名前の最後の空白は無視されます。
 - c) 大文字と小文字が区別されます。

区切付識別子は、大文字と小文字が区別された名前を好むユーザーにとって朗報となるかもしれませんが、一方で厄介な問題を抱えています。

標準 SQL では大文字と小文字は区別されないため、他の RDBMS で大文字と小文字を意識しないプログラミングになれてきた人々を戸惑わせることとなります。

ISQL では二重引用符を付けない名前に付いてはすべて大文字に変換してから実行されます。この辺は **IBConsole** や **IBOConsole** でも同様です。ただし **IBOConsole** では、**Dialect3** のデータベース使用時に、大文字に変換した上で、二重引用符が付加されて実行されてしまうようです。

そのため、**Delphi** の **dbExpress** で SQL 文を発行する際に小文字の名前がはねられてしまい、エラーが出るということが起こります。

筆者が確認したのは **Delphi 6** の **dbExpress** ですが、**SELECT** 文は小文字でも通るのに、**INSERT** 文は大文字で無いとだめという現象に出会い、原因を解明するのに手間取りました。

また、筆者が日本語化して公開している **IBOConsole** では、**GUI** を使用してテーブル等の作成ができるのですが、この場合そのまま二重引用符が付加されてしまうため、大文字小文字混在のオブジェクト名を使用すると、後になっていちいちすべて二重引用符で囲まないと SQL が通らないという事態に陥ってしまいます。

そこで、SQL については原則として大文字で書くということを徹底するに限るというのが結論です。ユーザーの自由度を高めるための設計が、逆にユーザーを縛っているような気がして残念です。

※ところで、本書ではサンプルの SQL を見やすくするためにオブジェクト名については大文字小文字混在で表記しています。上記主張とは逆になりますが、表現としてはやはりこちらの方がわかりやすいですね。

Appendix A Firebirdのデータ型

数値データ型

データ型	サイズ	範囲/有効桁数	説明
SMALLINT	16 ビット	-32,768 ~ 32,767	符号付短整数値型。
INTEGER	32 ビット	-2,147,483,648 ~ 2,147,483,647	符号付長整数値型。
FLOAT	32 ビット	$1.175 \times 10^{-38} \sim 3.402 \times 10^{38}$	IEEE 単精度浮動小数点型。有効桁数 7 桁。
DOUBLE PRECISION	64 ビット	$2.225 \times 10^{-308} \sim 1.797 \times 10^{308}$	IEEE 倍精度浮動小数点型。有効桁数 15 桁。
DECIMAL (precision, scale)	可変長 (16, 32 または 64 ビット)	precision=1~18。正確に格納される有効桁数を指定する。 scale=格納可能な小数点以下の桁数 (precision 以下でなければならない)	特定の小数点以下の桁数を持つ数値。
NUMERIC (precision, scale)	可変長 (16, 32 または 64 ビット)	precision=1~18。正確に格納される有効桁数を指定する。 scale=格納可能な小数点以下の桁数 (precision 以下でなければならない)	特定の小数点以下の桁数を持つ数値。

Firebirdのサポートする数値型は、16 ビット・32 ビットの整数型(INTEGER、SMALLINT)、単精度・倍精度の浮動小数点型(FLOAT、DOUBLE PRECISION)、書式付固定小数点型(DECIMAL、NUMERIC)となります。

整数データ型

FirebirdではINTEGERとSMALLINTの2つの整数型を使用することが出来ます。それぞれの有効桁数は上記の通りです。整数データ型では以下の演算を行うことが出来ます。

- 比較演算子(=, <, >, >=, <=)を使った比較を行えます。
- CONTAINING, STARTING WITH, LIKEのような他の演算子は、数値として文字列の比較を行います。
- 算術演算子を使って複数の整数での加減乗除を行えます。
- 複数のデータ型で算術演算を行うとき、FirebirdはINTEGER、FLOAT、CHARデータ型の間で自動的に型変換を行います。数値データを他のデータ型と比較する演算では、Firebirdはまずその他のデータ型を数値型に変換してから、比較を行います。
- テーブルのレコードは、SELECT文のORDER BY句に整数データ型を指定することで、降順または昇順にソートできます。

浮動小数点データ型

FirebirdにはFLOATとDOUBLE PRECISIONの2つの浮動小数点データ型があります。FLOATは32ビット単精度の浮動小数点となり、約7桁の有効桁数となります。DOUBLE PRECISIONは64ビット倍精度の浮動小数点で、約15桁の有効桁数となります。浮動小数点データ型においては、小数点以下の桁数は増減する(浮動)ので、同じ列に12.345と1.23という値を格納することが可能です。

固定小数点データ型

Firebirdは、通貨を扱う場合などに利用する固定小数点の数値データ型として、NUMERICとDECIMALの2つのSQLデータ型をサポートしています。どちらのデータ型でも有効桁数と小数点以下の桁数を指定することが出来ます。

- 有効桁数(precision)は、整数部と小数部をあわせた最大の桁数です。有効桁数として指定できる範囲は1~18となります。
- 小数点以下の桁数(scale)は、小数点より右側になる数値の桁数です。小数点以下の桁数に指定できる範囲は0~precisionまでとなります。scaleは必ずprecision以下でなければなりません。
- NUMERICとDECIMALの相違は、NUMERIC(p, s)が厳密にp桁が格納され、小数点以下の桁数が厳密にs桁となりますが、DECIMAL(p, s)ではp桁以上が格納され、小数点以下が厳密にs桁となります。
- NUMERICとDECIMALの2つの固定小数点データ型はダイアレクト1とダイアレクト3で、実際の格納方法が違ってきます。そのため、ダイアレクト1からダイアレクト3へデータベースを移行する際は一旦バックアップしてリストアするなどの手順が必要となります。

数値データ型の演算上の注意点

Firebirdでは、ダイアレクト1とダイアレクト3で、数値データ型の算術演算の結果が異なってきます。

- ダイアレクト1では二つの整数型数値または二つの固定小数点型数値の除算の商はDOUBLE PRECISION型の浮動小数点数値となります。
- ダイアレクト3では、除算の商の小数点以下の有効桁数は、除数と被除数のスケールの合計となります。したがって、二つの整数型数値の除算の結果は整数型数値となります。
- 上記の結果として、1/3を実行した場合、ダイアレクト1では0.333333333333333e0となりますが、ダイアレクト3では0となってしまいます。整数型数値同士の除算では注意してください。

APENDIX B
Firebird データ型の変換可能性一覧表

変換元	変換先											
	BLOB	CHAR	DATE	Dec.	Dbl.	Flot.	Int.	Num	Tstm	TIME	Sml.	Var.
BLOB												
CHAR		○										○
DATE		○	○						○			
Dec.		○		○				○				○
Dbl.		○			○	○						○
Flot.		○			○	○						○
Int.		○		○	○		○	○				○
Num		○						○				○
Tstm		○							○	○		
TIME		○							○	○		
Sml.		○		○	○	○	○	○			○	○
Var.		○										○

※各データ型はそれぞれ以下のように省略してあります。Dec. =DECIMAL、Dbl. =DOUBLE、Flot. =FLOAT、Int. =INTEGER、
Num =NUMERIC、Tstm =TIMESTAMP、Sml. =SMALLINT、Var. =VARCHAR

※左端の列で変換元のデータ型を特定して、右へたどると変換可能なデータ型の列に○をつけてあります。