

Yet another Open Source Database

Firebird RDBMS 傾向と対策 3

林 務 (株式会社アペックス)
mail : hayashi@apex-jp.com

はじめに

本書は、PS ネットワークより平成 15 年 8 月 5 日に初版が出版された『Pocket Tech Note Firebird RDBMS 傾向と対策 3』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「エラーコード一覧」と「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 IBConsole 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思いから、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 2」に引き続いて、オープンソースの RDBMS、Firebird のストアードプロシージャとトリガーに関する解説をさせていただきます。前著では Firebird の SQL に関して、網羅的なリファレンスを提供することを目的として、逐条的な解説をおこないました。また、Firebird のトランザクションについて、2つの並行動作するトランザクションを実際に操作してみることを通して解説を試みました。

本書では、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たいと思います。読者のみなさんが、業務においてトリガーやストアードプロシージャを作成される際の、一助となれば幸いです。

商用のデータベース・システムでは一般的なトリガー/ストアードプロシージャですが、オープンソースのデータベースでも PostgreSQL が強力な環境を提供していますが、Firebird も InterBase の機能を引き継いでいますので、まげず劣らず柔軟な機能を提供しています。

また、触りだけとなりますが、ユーザー定義関数 (UDF) によって機能を拡張することが容易なので、トリガー/ストアードプロシージャと組み合わせることによって、開発者の自由度は無限に広がります。UDF の詳細については、別稿にて解説をさせて頂く予定となっております。

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。また、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。世界中で Firebird を支えているプロジェクトの皆さん、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 3	1
PSQL - FIREBIRD のプロシージャ・トリガー言語	4
FIREBIRD のオブジェクト命名規則	5
I/BOCONSOLE の利用方法	6
データベースの作成	6
XLIBUDF の利用方法 - WRITEDeBUG 関数	12
SQL リファレンス-プロシージャ・トリガー関連	14
CREATE PROCEDURE	14
ALTER PROCEDURE	15
DROP PROCEDURE	15
CREATE TRIGGER	16
ALTER TRIGGER	16
DROP TRIGGER	17
CREATE EXCEPTION	17
ALTER EXCEPTION	17
DROP EXCEPTION	17
PSQL リファレンス	18
DECLARE VARIABLE	18
FOR SELECT ... DO	18
IF ... THEN ... ELSE	19
EXECUTE PROCEDURE	20
WHILE ... DO	21
EXIT	22
SUSPEND	22
POST_EVENT	23
EXCEPTION	23
トリガーで使用できるコンテキスト変数	24
NEW コンテキスト変数	24
OLD コンテキスト変数	25
WHEN ... DO	25
例外処理	26
SQL エラーの処理	26
Firebird エラーコードの処理	26
例外処理の例	26
ストアードプロシージャの利用例	28
サンプルデータの作成	28
UDF の作成	28
サンプルデータ追加用のストアードプロシージャの作成	29
クロス集計ストアードプロシージャ(1)	30
クロス集計ストアードプロシージャ(2)	31
クロス集計ストアードプロシージャ(3)	33
クロス集計ストアードプロシージャ(4)	35
指定した一連番号を返すストアードプロシージャ	37
トリガーの使用例	40
ジェネレータを使用したオートインクリメント	40
トリガーによるロギング	40
ビュートリガーによる更新制御	41
APPENDIX A FIREBIRD のデータ型	44
数値データ型	44
整数データ型	44
浮動小数点データ型	44
固定小数点データ型	44
数値データ型の演算上の注意点	44

文字データ型.....	45
日付時刻型.....	45
BLOB データ型.....	45

PSQL - Firebirdのプロシーチャー・トリガー言語

Firebirdのトリガー・プロシーチャー言語はPSQLと呼ばれ、OracleにおけるPL/SQLやMS SQL ServerにおけるTransact-SQLと同様に、データベースへの一連の処理をデータベースサーバー側に保存し、サーバーサイドでの実行を可能とするために使用されるプログラミング言語です¹。

SQLは通常プログラマが使用している、Java / C++ / Basic / Delphi / Fortran / Cobol などの3GL(第3世代言語)と呼ばれる手続き型の言語と違い、集合を対象とする非手続き型の言語です。PSQLはSQLを拡張したものではありますが、前者の手続き型言語に属するものです。

ANSI SQL99ではストアードプロシーチャーとトリガーに関する定義が行われています。ストアードプロシーチャーに関しては、SQL/PSM(Persistent Stored Module 永続格納モジュール)という規格に、トリガーについてはActive Databaseという規格にそれぞれまとめられました。

とはいえ、各RDBMSはそれぞれ独自にこうした機能を実装してきたため、実際のところ交換性はほとんど無いのが現状です。しかし、こうした実情にあわせて交換性がないからとストアードやトリガーを使用しないのであれば、RDBMSの能力を半分しか使用しないことになってしまいます。

ストアードプロシーチャー/トリガーを使用するときによくいわれることですが、利点としては以下のようなことがあげられます。

- ・ サーバーサイドでの実行なのでネットワークのトラフィックを減少させる
- ・ 一般にサーバーの方が能力が高いため処理が高速
- ・ RDBMS内で処理が完結するのでプロセス間通信なども減少し高速でメモリ効率がよい

もちろん、ストアードやトリガーも万能ではありませんし、クライアントサイドで処理を行う方がより適していることも多々あると思います。筆者としては、適材適所で、必要に応じて使い分ければいいのではないかと考えています。Oracleのストアードが流行した結果、メンテナンスの困難なシステムが残ってしまい苦労しているという話も聞きます。使い分けを誤らなければ、こんなに便利なものはないので、上手に利用したいものです。

さて、PSQLでは、通常のSQL構文に加えてIF... THEN... ELSE...文、FOR SELECT... DO文、WHILE... DO文等の制御構造が使用可能であり、SQLでは不可能であったり、可能であっても複雑になりすぎる処理を手続き的に行うことで、より容易に実装することが可能となっています。例外処理や、イベント通知などの処理も可能なので、より高度な操作を行うことができます。

Firebirdの独自性としては、テーブルとして使用可能なストアードプロシーチャーを作成可能という点をあげることができます。この機能の最大の利点は、どんな複雑な条件であっても手続き的に表現可能であれば、SELECT文に対して結果セットを返すことが可能だということにあります。また、ストアードプロシーチャーの出力パラメータをSELECT文の結果として返すことも可能なので、デバッグに利用することも可能です。

ただし、ストアードプロシーチャーの内部では動的にSQLを作成することはできないので、あらかじめ選択するテーブルや列を指定しておかなくてはならないという欠点もあります。その他、Firebirdでは、SELECT文のFROM句にサブクエリーを使用することができないので、代替として表を返すストアードプロシーチャーを使用するという使われ方をすることもあります。

さて様々な利点のあるストアードプロシーチャーですが、ご多分に漏れず他のRDBMSと同様、デバッグ環境が貧弱で、言語仕様としても単純な手続き型言語でしかありませんので、あまり複雑なプログラムを作成することは難しいかも知れません。継承や多態などのオブジェクト指向の要素は全く取り込まれていません。そのため、似たようなプログラムをいくつも作る羽目になるということも多々あります。繰り返しになりますが、使い方をよく考えて効率的に使用するようにしたいものです。

なお、本書では動作確認にfirebird-win32 1.0.2-Release (Firebird-1.0.2.908-Win32.exe)をWindows2000にインストールして利用し、標準で付属するisql.exeと筆者が日本語化を行っているオープンソースのGUI管理コンソールIBOConsole (Lorenzo Mengoni氏作)によって行いました。Firebirdプロジェクトのホームページと(株)アペックスのホームページからダウンロードして利用してください。

なお、IBOConsoleではisql.exeと若干動作が異なる場所が有りますので、注意してください。もちろん、Firebird自体の動作が変わるわけではありません。

デバッグ環境として利用する場合、IBOConsoleは、入力パラメータを設定してストアードを実行することができるので強力なサポートを提供します。また、SoftComplete Development社によるフリーのUDFライブラリxlibUDFを使用することによって、ストアードやトリガーからデバッグ用コンソールにメッセージを送信することができるので、デバッグの際には大変便利です。これらの用法については、後述いたします。

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。

¹ PL/SQLやTransact-SQLはそれ以外の拡張機能も提供しています。

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Delphi6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。

IBOConsole の利用方法

本書では、Lorenzo Mengoni 氏作のオープンソースのIGUI 管理コンソールである IBOConsole を利用して、ストアドプロシージャ・トリガーの作成及びテストを行います。Lorenzo 氏のホームページで配布されているものは、そのままでは日本語が文字化けしてしまいますので、

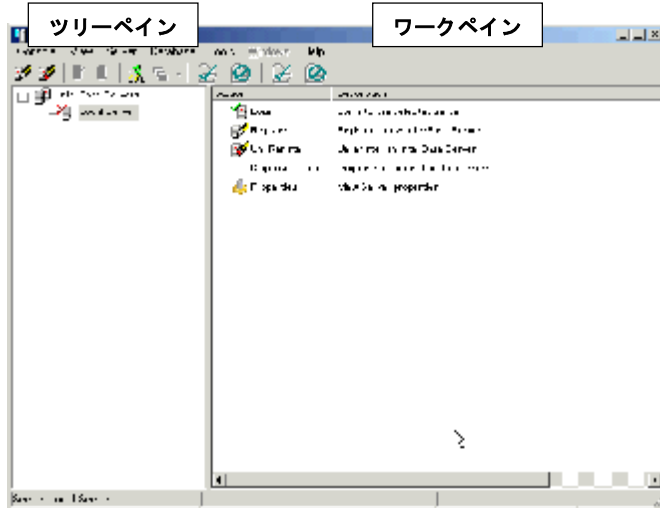
筆者が独自に日本語化（メニューなどは英語のままです）したものを（株）アペックスのホームページにて配布しています。本書で利用しているのは、この日本語対応版ですので、ダウンロードしてお使いください。

Lorenzo Mengoni 氏のホームページ

<http://www.mengoni.it/downloads.html>

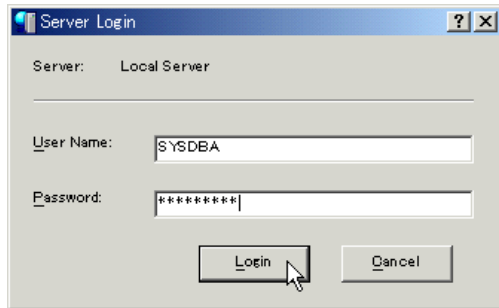
株式会社アペックスのホームページ

http://www.apex-jp.com/business_dev.html

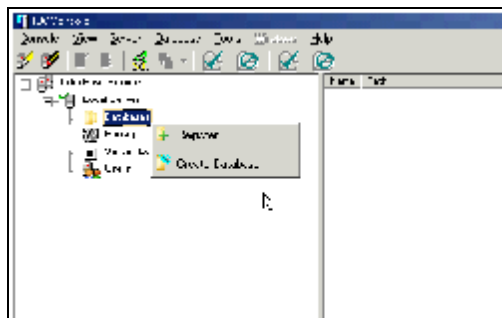


データベースの作成

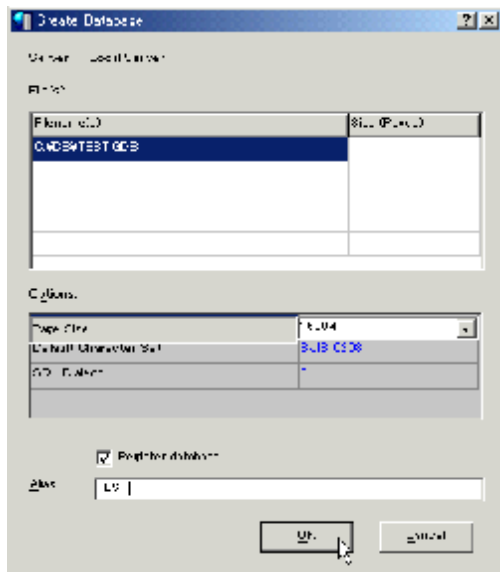
IBOConsole を起動すると、前ページの図にあるように、左側にツリーペインが右側にワークペインが表示されます。まずはローカルサーバーにログインするためにツリーペインの「Local Server」をダブルクリックして、デフォルトユーザーのユーザー名：SYSDBA・パスワード：masterkey でログインして下さい。もちろん、他のユーザー/パスワードが使用できる場合はそれらで構いません。



ログインに成功すると以下のような内容がツリーペインに表示されます。一つ一つの要素をノードと呼びます。この状態で、「Databases」ノードを右クリックして「Create Database」を実行します。



すると、データベースの作成ダイアログが表示されます。



ここで指定できるのは、以下の要素です。

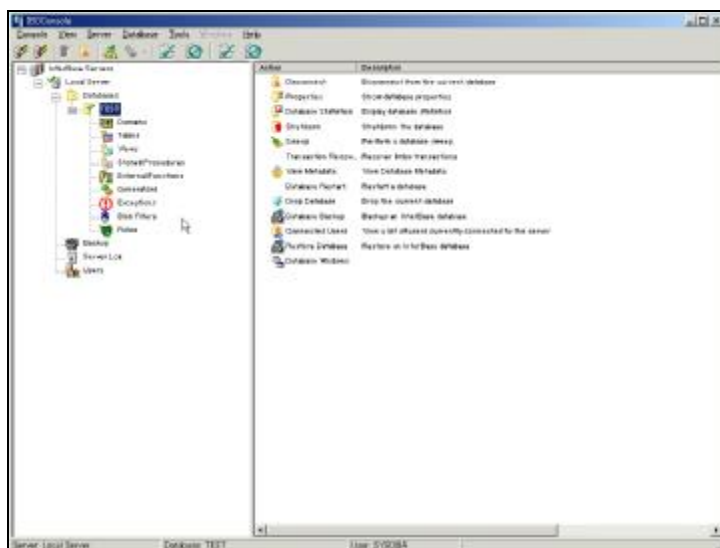
- データベースのファイル名とサイズ (Page 数)
- ページサイズ
- デフォルトキャラクタセット
- SQL ダイアレクト

データベースファイルは複数指定することが可能で、その場合にはそれぞれのファイルのサイズを指定することに意味が出てきます。今回はテスト用データベースですから、シングルファイルで十分です。最後のファイルにおけるサイズ指定は無視されて、OS の上限まで拡張されるので、これも指定しなくて良いでしょう。適当なパスに **TEST.GDB** というファイル名を指定します。

ページサイズには、**1024/2048/4096/8192/16384** の各数値を一覧から指定します。一般に大きい数値を指定した方がパフォーマンスが上がりますので、ここは **16384** を指定しておきます。トレードオフとしてファイルの利用効率が下がる可能性があります、気にする必要はないでしょう。

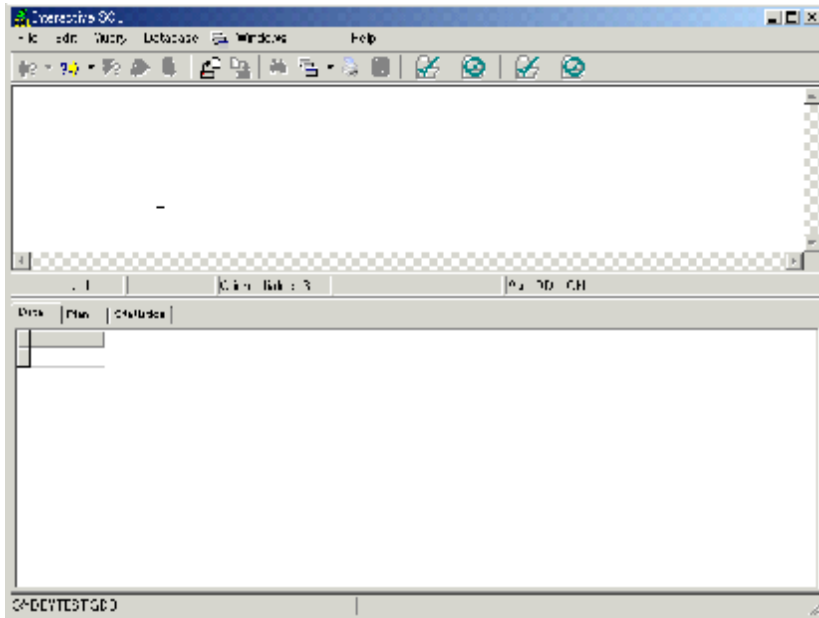
デフォルトキャラクタセットは、**Windows** の場合は **SJIS_0208** を **Linux** の場合は **EUCJ_0208** を指定しておきましょう。**SQL** ダイアレクトは新しくデータベースを作成する場合には、**3** を指定しておきます。

「データベースを登録する」のチェックボックスをつけたままにして、「エイリアス」に **TEST** と入力し、**OK** すればデータベースが作成されます。

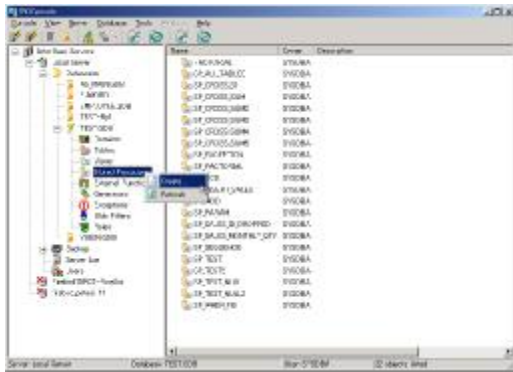


「Databases」ノードの下に「TEST」エイリアスが登録されています。その下にさらに、各データベースオブジェクトを示すノードが表示されています。

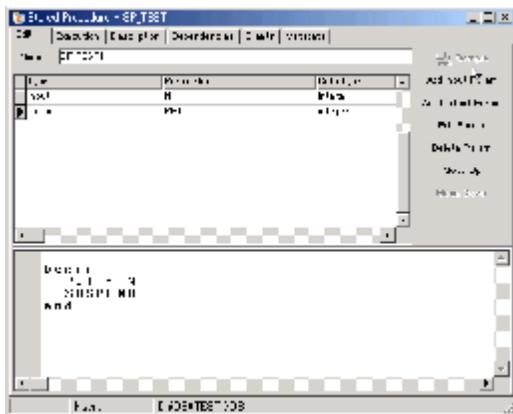
この状態で **Interactive SQL** をツールバーのボタンから起動すると、**TEST** データベースに接続した状態で **SQL** 文を実行できるようになります。



Interactive SQL は別ウィンドウで起動します。上半分が SQL スクリプトの入力部で、**SELECT** 文の結果は、下側のグリッドに表示されます。**Plan** タブには、クエリーの実行に際して使用されたアクセスパスが表示されます。**Statistics** タブには実行に要した時間やメモリなどの統計情報が表示されます。



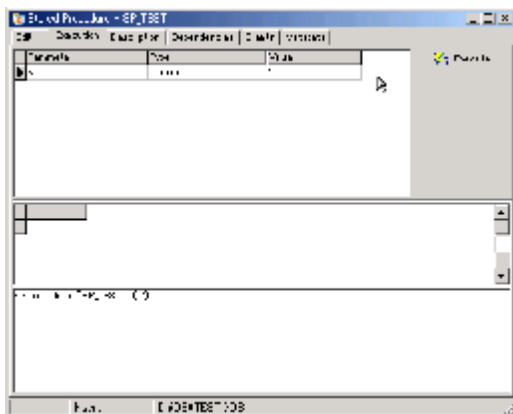
ストアドプロシージャを作成するには、ツリーペインで「Stored Procedure」を右クリックするか、「Stored Procedure」を選択した状態でワークベンチ上で右クリックをし、「Create」メニューを実行するとストアド・プロシージャ・エディタが表示されます。



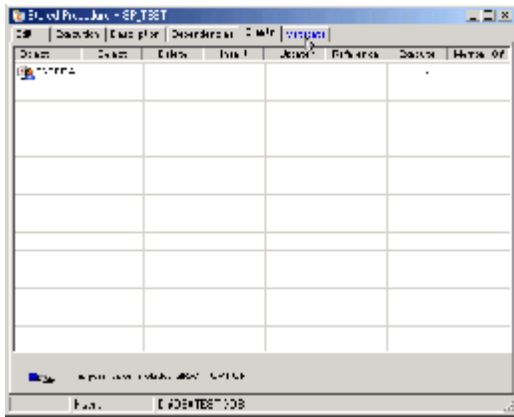
ストアド・プロシージャ・エディタの画面です。プロシージャ名、入力/出力パラメータの追加/削除と、本文の記述・編集ができます。何らかの変更を加えると、右上のコンパイルボタンが有効となるので、変更をデータベースに格納するためには、このコンパイルボタンをクリックしてください。

最初に起動したときは、コンパイルボタンをクリックしてエラーが発生しないで、プロシージャが作成されてからでないと、他のタブをクリックして表示させることはできません。

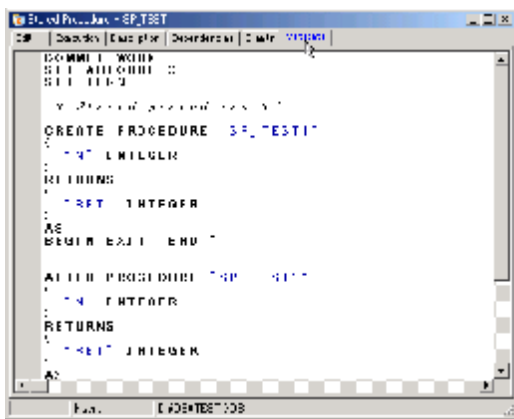
ここで記述する本文は、BEGINで始まり、ENDで終了するPSQL文ですが、最後のターミネータは必要ありません。



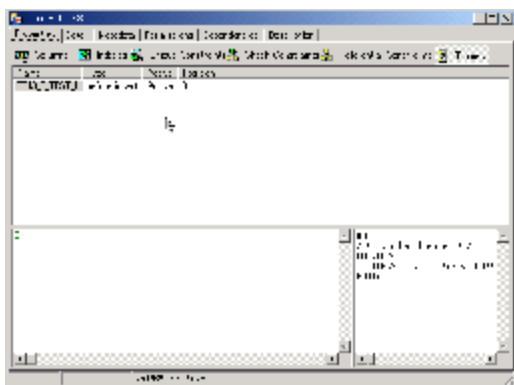
ストアド・プロシージャ・エディタの「Execute」画面です。ここで、(もしあれば) 入力パラメータを指定して、Execute ボタンをクリックすると下側のグリッドに結果が表示されます。ただし、一番下側のスクリプト・エリアを見ているとわかるのですが、あくまでも SELECT 文の結果として表示しているため、SUSPEND を使用しない実行型プロシージャについては結果を表示することができません。



ストアド・プロシージャ・エディタの「Grants」画面です。ここでは、このプロシージャに関する SQL 特権を確認することができます。追加/削除/変更はできないので、Interactive SQL や isql.exe を利用してください。テーブルの SQL 特権は GUI から変更できるので、ここも今後改良していかなくてはならないところでしょう。



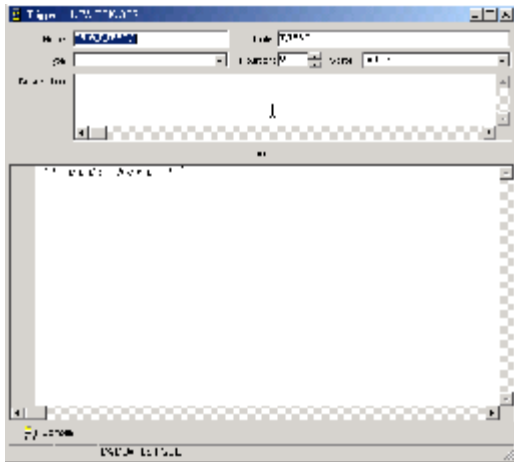
ストアド・プロシージャ・エディタの「Metadata」画面です。ここでは、ストアド・プロシージャのメタデータを参照することができます。ただし、IBObjects の特性なのか、CREATE 文と ALTER 文を組み合わせたものとなっています。また、入力/出力パラメータの括弧の前後で改行が入っているため、これをそのまま Interactive SQL にカット・アンド・ペーストすると実行できないという不具合がありますので、注意してください。



トリガーを GUI から作成する場合は、IBOConsole のテーブル・エディタを表示して、「Properties」タブから、「Trigger」ボタンをクリックしてトリガー一覧を表示させます。

上部にトリガーの一覧、下側の左半分には選択したトリガーのポジションが、右側にはトリガー本文が表示されます。ポジションは一覧にも表示されているので、この表示も今後改善した方が良いでしょう。

ここで、トリガー一覧上で右クリックするとコンテキスト・メニューが表示されるので、Create/Edit/Drop をそれぞれ実行します。当然、一つもトリガーのない状態では Create しか実行できません



トリガー・エディタです。トリガー名を指定して、タイプを選択し、ポジションを数値で指定し、「State」を **Active/Inactive** から指定して、本文を記述します。

テーブル名は変更することができません。必要なテーブルのプロパティを表示させてから、そのテーブルのトリガーを作成するようにしてください。

「Description」にはトリガーの内容をコメントとして記述しておくことができます。

本文には、**BEGIN** から始まって **END** で終了する **PSQL** 文を記述しますが、最後にターミネータを付ける必要はありません。すべてを指定し、本文を記述したら左下のコンパイルボタンをクリックして、変更をデータベースに反映してください。

xlibUDF の利用方法 - WRITEDeBUG 関数

xlibUDF は、SoftComplete Development 社が配布しているフリーの UDF ライブラリです。残念ながら、ソースコードは公開されていないので、Windows 以外の環境で利用することはできません。もっとも、文字列関係の関数などは他の UDF でも提供されているので困ることはあまりないと思います。xlibUDF は以下のロケーションから入手してください。

SoftComplete Development 社ダウンロードページ
<http://www.softcomplete.com/download.asp>

ダウンロードした xlibudf.zip を解凍すると以下のファイルが展開されます。

```
html doc\img\ibdbgscreen.gif
html doc\img\logo3.gif
html doc\index.html
html doc\xlibfunc.html
IBDebugWnd.exe
xlibudf.dll
xlibudf.sql
```

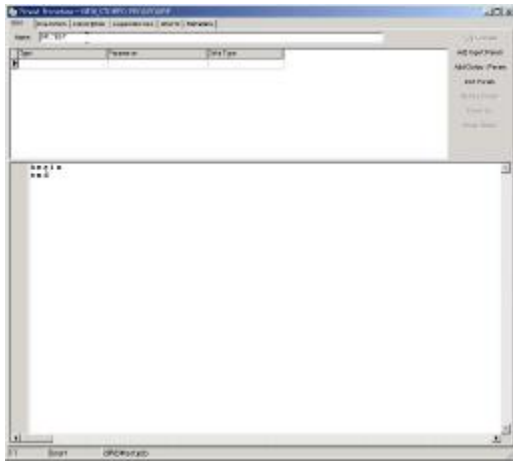
xlibudf.dll が本体なので、これを Firebird のインストールフォルダ直下の UDF フォルダにコピーしておきます。デフォルトでは、C:\Windows\ProgramFiles\Firebird\UDF になります。また、IBDebugWnd.exe はそれ自体が DCOM サーバーなので、適当なフォルダにコピーした後、一度実行することでレジストリに登録されますので、とにかく一度実行してください。

今回は、ストアードトリガーのデバッグ用途に WRITEDeBUG() 関数を利用してみます。WRITEDeBUG() 関数はその名の通り、デバッグ用コンソールに文字列を書き出すための関数です。DECLARE FUNCTION 文での定義は以下の通りです。

```
/* function WriteDebug(FileName, Str: pchar): integer;
*/
DECLARE EXTERNAL FUNCTION WriteDebug
    CSTRING(255)
    RETURNS INTEGER BY VALUE
    ENTRY_POINT WriteDebug MODULE_NAME XLIBUDF.DLL;
```

では、実際に使ってみましょう。まず、上記の DECLARE FUNCTION 文を IBOConsole の Interactive SQL で実行してください。IBOConsole では、左側のツリーペインで「External Functions」を選択して、右クリックのコンテキストメニューから「Create」を実行すると UDF を登録することもできますが、UDF ライブラリには登録用の SQL スクリプトが付属しているのでこれを実行する方が手取り早く簡単です。

次に、WRITEDeBUG を利用した最も簡単なストアードを作成してみます。まず、IBOConsole でツリーペインから「Stored Procedure」を選択し、右クリックメニューから「Create」を実行します。



「Stored Procedure - NEW STORED PROCEDURE」ウィンドウが表示されるので、「Name」欄に「SP_TEST」と入力します。

次に、右端の「Add Input Param」をクリックして、入力パラメータを設定します。ここでは、MES という名前で CHAR(127)のサイズとしましたが、データベースのデフォルトキャラクタセットを SJIS_0208 とした関係上、1 文字を 2 バイト換算で登録されるため、仕方なくこうしました。WRITEDeBUG() 関数自体は、CSTRING(255)のパラメータを受け取るので、ASCII 等の 1 バイトのキャラクタセットで CHAR(255)にするのがよいのですが、残念ながら IBOConsole では、ストアードのパラメータ作成に際してキャラクタセットを指定することができません。必要な場合は、Interactive SQL や isql.exe を利用して SQL スクリプトからストアードを作成する必要があります。

同様に、「Add Output Param」をクリックして、出力パラメータを設定します。ここでは、RET という名前で INTEGER 型のパラメ

ータを作成しました。

最後に、本文を記述します。下側のエディット領域で **begin** と **end** の間に本文を記述していくことになります。ここでは、以下のよう
に記述しました。

```
begin  
  RET = WRITTEDEBUG(MES);  
  SUSPEND;  
end
```

たったこれだけです。本文を記述すると、右上の「**Compile**」ボタンが使用可能になりますので、クリックしてストアデータをデータベースに登録します。コンパイルエラーが出た場合は、もう一度見直してください。うまくいけば、コンパイルボタンは再び使用不可になり、ユーザーは上部のタブを使用して、ストアのテスト実行などを行うことができますようになります。

SQL リファレンス・プロシージャ・トリガー関連

PSQL の解説に入る前に、プロシージャ・トリガーに関連する SQL 文のリファレンスを以下に示すことにします。

プロシージャ・トリガーの本文は、PSQL によって記述しますが、データベースへの定義/更新/削除は SQL 文として行います。

PSQL の行末をあらわす「;」と isql のターミネータの規定値が同じなので、これらの SQL 文の前でターミネータを変更しておかなくてはなりません。通常、プロシージャ・トリガー関連の SQL 文の前後で SET TERM 文を実行して、ターミネータを変更・復旧しています。注意して下さい。

CREATE PROCEDURE

CREATE PROCEDURE 文は、現在のデータベースに新しいストアードプロシージャを作成します。isql.exe や IBOConsole の Interactive SQL で実行が可能です。ただし、ストアードプロシージャの各文は「;」を文の区切として使用するので、上記の環境で実行するためには SET TERM 文を利用してターミネータを一時的に変更する必要があります。

```
CREATE PROCEDURE プロシージャ名
[( 入力パラメータ名 <データ型>
  [, 入力パラメータ名 <データ型> ...])]
[RETURNS 出力パラメータ名 <データ型>
  [, 出力パラメータ名 <データ型> ...]]]
AS <プロシージャ本文> [ ターミネータ ]
<プロシージャ本文> =
[ <ローカル変数宣言リスト> ] <ブロック>
<ローカル変数宣言リスト> =
DECLARE VARIABLE 変数名 <データ型>;
[DECLARE VARIABLE 変数名 <データ型>; ...]
<ブロック> =}
BEGIN
<複文>
[<複文> ...]
END
<複文> = <ブロック> | ステートメント ;
<データ型> =
SMALLINT |
INTEGER |
FLOAT |
DOUBLE PRECISION |
{DECIMAL | NUMERIC} [( 有効桁数 [, 精度 ])] |
{DATE | TIME | TIMESTAMP} |
{CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
[( 文字数) [CHARACTER SET キャラクタセット名 ] |
{NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
[VARYING] [( int )]
```

- ・ プロシージャ名には、各データベースに存在するテーブル/ビュー/プロシージャに対して一意な名前ではなくてはなりません。
- ・ 入力/出力パラメータのデータ型については、APENDIX A を参照してください。
- ・ ターミネータは、前述した isql や Interactive SQL において SET TERM 文で指定したターミネータを記述します。isql や Interactive SQL ではここで記述したターミネータまでを 1 つの SQL 文として扱うため、このような措置が必要となります。IBOConsole の GUI を利用してストアードプロシージャを作成する場合などには、必要ありません。
- ・ ローカル変数は「;」で区切って必要なだけ宣言することができます。ここで宣言したローカル変数は、このプロシージャ内でのみ有効です。
- ・ ブロックは BEGIN ではじまり、END までの 1 つ以上のステートメントで構成されます。各ステートメントは「;」までが 1 行となりますが、END の後に「;」は必要ありません。Pascal の文法とはちょっと違うので注意が必要です。
- ・ WRITDEBUG() 関数の使用法で示したストアードプロシージャを CREATE PROCEDURE 文で記述すると以下のようになります。

```
SET TERM ^ ;
```

```
CREATE PROCEDURE SP_TEST
( MES CHAR(255) CHARACTER SET ASCII )
RETURNS
( RET INTEGER )
AS
BEGIN
  RET = WRITDEBUG(MES);
  SUSPEND;
END
```

```
^  
SET TERM ; ^
```

※この例では、ターミネータを「;」から「^」に変更して **CREATE PROCEDURE** 文を記述しています。

- **SUSPEND** 文については後述しますが、選択型プロシージャとして、**SELECT** 文でストアードプロシージャを扱うために必要な構文です。

ALTER PROCEDURE

ALTER PROCEDURE 文は、**CREATE PROCEDURE** 文によってすでに作成されているストアード・プロシージャを変更するために使用します。

ALTER PROCEDURE プロシージャ名

```
[(入力パラメータ <データ型>  
[, 入力パラメータ <データ型> ...])]  
[RETURNS (出力パラメータ <データ型>  
[, 出力パラメータ <データ型> ...])]  
AS <プロシージャ本文> [ターミネータ]
```

- **ALTER PROCEDURE** 文によって、入力/出力パラメータとプロシージャの本文を変更することができます。ただし、どれか一つだけを変更する場合でも、すべての情報を記述する必要があります。つまり、基本的に **CREATE PROCEDURE** 文を書き換えて再実行するようなことになります。
- **ALTER PROCEDURE** 文の構文は、**CREATE PROCEDURE** と同一ですので、**CREATE PROCEDURE** 文の解説を参照してください。
- ストアード・プロシージャの変更ができるのは、作成者/SYSDBA 及び、UNIX/Linux の root 権限をもつユーザーだけです。

DROP PROCEDURE

DROP PROCEDURE 文は、すでに作成されているストアード・プロシージャをデータベースから削除します。

DROP PROCEDURE プロシージャ名;

- 他のプロシージャ/ビュー/トリガーで使用されているプロシージャを削除することはできません。
- `isql.exe` の **SHOW PROCEDURE** コマンドを使用すると、ストアード・プロシージャの依存関係を表示することができます。この使用法の場合は、プロシージャ名を指定しないで **SHOW PROCEDURE** コマンドを実行してください。

```
SQL> show procedure;
```

```
Procedure Name Dependency, Type
```

```
=====
```

```
SP_ALL_TABLES RDB$RELATIONS, Table
```

```
SP_CROSS_SUM T_CROSSDAY, Table
```

```
T_SALES, Table
```

```
EXP_INVALID_MONTH, Exception
```

```
EXP_INVALID_YEAR, Exception
```

- ストアード・プロシージャを削除できるのは、作成者/SYSDBA 及び、UNIX/Linux の root 権限をもつユーザーだけです。

CREATE TRIGGER

CREATE TRIGGER 文は、現在のデータベースに新しいトリガーを作成します。isql.exe や IBOConsole の Interactive SQL で実行が可能です。ただし、ストアードプロシージャと同様に、トリガーの各文は「;」を文の区切として使用するので、上記の環境で実行するためには SET TERM 文を利用してターミネータを一時的に変更する必要があります。

```
CREATE TRIGGER トリガー名 FOR テーブル名
  [ACTIVE | INACTIVE]
  {BEFORE | AFTER}
  {DELETE | INSERT | UPDATE}
  [POSITION 数値]
  AS <トリガー本文> ターミネータ
<トリガー本文> = [<変数宣言リスト>] <ブロック>
<変数宣言リスト> =
  DECLARE VARIABLE 変数 <データ型>;
  [DECLARE VARIABLE 変数 <データ型>; ...]
<ブロック> =
  BEGIN
  <複文>
  [<複文> ...]
  END
<複文> = <ブロック> | ステートメント;
<データ型> =
  SMALLINT |
  INTEGER |
  FLOAT |
  DOUBLE PRECISION |
  {DECIMAL | NUMERIC} [( 有効桁数 [, 精度 ] )] |
  {DATE | TIME | TIMESTAMP} |
  {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
  [(文字数)] [CHARACTER SET キャラクタセット名 ] |
  {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
  [VARYING] [( int )]
```

- トリガー名には、各データベースにおいて一意な名前ではなくてもなりません。テーブル名やプロシージャ名と同一でも問題はありませんが、同じ名前のトリガーを作成することはできません。
- 入力/出力パラメータのデータ型については、**APPENDIX A**を参照してください。
- ターミネータは、前述した isql や Interactive SQL において SET TERM 文で指定したターミネータを記述します。isql や Interactive SQL ではここで記述したターミネータまでを1つの SQL 文として扱うため、このような措置が必要となります。IBOConsole の GUI を利用してトリガーを作成する場合などには、必要ありません。
- ローカル変数は「;」で区切って必要なだけ宣言することができます。ここで宣言したローカル変数は、このプロシージャ内でのみ有効です。
- ブロックは BEGIN ではじまり、END までの1つ以上のステートメントで構成されます。各ステートメントは「;」までが1行となりますが、END の後に「;」は必要ありません。Pascal の文法とはちよつと違うので注意が必要です。
- ACTIVE と INACTIVE は、それぞれトリガーを有効/無効にするために指定します。トリガーの作成後は、ALTER TRIGGER 文で変更することが可能です。
- BEFORE/AFTER と DELETE/INSERT/UPDATE はそれぞれを組み合わせて指定します。例えば、INSERT の実行前に値をチェックしたい場合は、BEFORE INSERT を指定するなどします。
- POSITION 句は、指定した順序でトリガーを起動するためのものです。0～32676 の範囲の値を指定できます。値の小さいものほど早く起動され、規定値は最小の 0 となります。
注意しなくてはならないのは、同一のコンテキストで起動する複数のトリガーに同一の値を指定した場合に、起動順序が不定になるということです。一つのテーブルに多数のトリガーを設定することはそうないと思いますが、注意してください。

ALTER TRIGGER

ALTER TRIGGER 文は CREATE TRIGGER 文によって、既に作成されているトリガーを変更するために使用します。

```
ALTER TRIGGER トリガー名
  [ACTIVE | INACTIVE]
  [{BEFORE | AFTER} {DELETE | INSERT | UPDATE}]
  [POSITION number]
  [AS <トリガー本文>] [ターミネータ]
```

- ・ **ALTER TRIGGER** 文は、**ALTER PROCEDURE** 文と違い、トリガーのヘッダ情報と本文をそれぞれ別々に変更することができます。また、両方同時に変更することもできます。トリガーの名前を変更することはできません。
- ・ トリガーの起動状態を **ACTIVE** から **INACTIVE** に変更するだけという使用方法も可能です。

ALTER TRIGGER TRG_TEST_BI INACTIVE;

- ・ **BEFORE INSERT** から **AFTER INSERT** への変更のように、起動イベントの変更も可能です。
- ・ トリガーの変更ができるのは、作成者/**SYSDBA** 及び、**UNIX/Linux** の **root** 権限をもつユーザーだけです。

DROP TRIGGER

DROP TRIGGER 文は、すでに作成されているトリガーをデータベースから削除します。

DROP TRIGGER トリガー名;

- ・ **DROP TRIGGER** 文で削除できるのは、ユーザーによって作成されたトリガーだけとなります。**CHECK** 制約に伴って、システムが作成したトリガーを **DROP TRIGGER** 文で削除することはできません。**ALTER TABLE** 文で **CHECK** 制約を変更/削除するなどしてください。

トリガーを削除できるのは、作成者/**SYSDBA** 及び、**UNIX/Linux** の **root** 権限をもつユーザーだけです。

CREATE EXCEPTION

CREATE EXCEPTION 文は、**PSQL** の **EXCEPTION** 文で使用する例外を定義します。

CREATE EXCEPTION 例外名 '例外メッセージ';

- ・ ここで定義できるのは、例外名と対応するメッセージだけです。

ALTER EXCEPTION

ALTER EXCEPTION 文は、すでに定義されている例外のメッセージを変更します。

ALTER 例外名 '例外メッセージ';

- ・ 例外を変更できるのは、作成者/**SYSDBA** 及び、**UNIX/Linux** の **root** 権限をもつユーザーだけです。

DROP EXCEPTION

DROP EXCEPTION 文は、すでに定義されている例外を削除します。

DROP EXCEPTION 例外名;

- ・ 例外を削除できるのは、作成者/**SYSDBA** 及び、**UNIX/Linux** の **root** 権限をもつユーザーだけです。

PSQL リファレンス

本章では、PSQL の構文について順を追って解説していきます。PSQL はストアードプロシージャとトリガーの両方で基本的に共通となっています。

SQL は DML 文として、INSERT、UPDATE、DELETE および単一行を返す SELECT 文が実行可能です。複数行を返す SELECT 文は FOR SELECT ... DO 文で扱います。

変数の宣言と使用、代入文、制御構文として IF ... THEN ... ELSE 文や WHILE ... DO 文、コンテキスト変数、イベントの通知や例外処理などの拡張構文が使用できます。

このうち、OLD と NEW のコンテキスト変数はトリガーでのみ使用することができます。逆に入力・出力パラメータ、SUSPEND 文と EXIT 文はストアードプロシージャでしか使用することができません。

また、以下の SQL 構文はストアードプロシージャとトリガーでは使用することができません。

- SQL データ定義文 (DDL) : CREATE、ALTER、DROP、DECLARE EXTERNAL FUNCTION、DECLARE FILTER 文
- トランザクション制御文 : SET TRANSACTION、COMMIT、ROLLBACK の各文
- CONNECT、DISCONNECT 文及び他のデータベースへの SQL 文の送信
- GRANT と REVOKE の各文
- SET GENERATOR 文
- その他の gpre や DSQL で使用可能な構文も使用することができませんが、本書では詳細を割愛します

Oracle の PL/SQL 等と比べると構文の数はかなり少ないので、覚えるのにそれほど時間はかからないでしょう。なお、Firebird の PSQL にはカーソル関連の機能はありません。そのため、複数の行を取得して、前後に移動しながらデータを操作することはできません。とはいえ、PL/SQL でもカーソル FOR ループ文を使用することでカーソル変数を利用したプログラムをほぼ完全に置き換え可能といわれていますから、実用上の問題は無いと思います。

ただし、Oracle ではカーソル For ループ文を使用する場合、宣言したカーソルに合わせて暗黙にレコード型の変数が準備されるため、それぞれの列に対して一つ一つのデータ型を指定する必要がないという利点があります。Firebird では、一つ一つの列に対して、必要なだけデータを格納する変数を宣言しなくてはいけないため、大規模なテーブルに対する操作は面倒になるでしょう。

なお、本書では SELECT 文でテーブルの代わりに使用可能なストアードプロシージャを「**選択型プロシージャ**」(SELECT PROCEDURE) と呼び、その他の関数または手続きとして使用可能なストアードプロシージャを「**実行型プロシージャ**」(EXECUTABLE PROCEDURE) と呼ぶことにします。

DECLARE VARIABLE

トリガーとストアードプロシージャ内で利用する、ローカル変数を宣言します。

DECLARE VARIABLE 変数名 データ型;

- 変数名は各トリガー/ストアードプロシージャ内で一意でなくてはなりません。変数のスコープは、各トリガー/ストアードプロシージャに限定されます。
- データ型には Firebird で有効なデータ型を指定できますが、BLOB 及び配列については指定することができません。
- それぞれのローカル変数毎に DECLARE VARIABLE で始まり、「;」で終わります。
- ローカル変数の宣言位置は、プロシージャの宣言と本文の間になります。

```
CREATE PROCEDURE SP_TEST2
AS
  DECLARE VARIABLE RET INTEGER;
BEGIN
  . . .
```

FOR SELECT ... DO

FOR SELECT ... DO 文は、指定して SELECT 文によって抽出された結果の各行に対して、DO の後で指定するブロック又は PSQL 文を繰り返し実行します。この文は、トリガー/ストアードプロシージャの両方で使用することができます。

FOR <SELECT 表現> DO <ブロック | ステートメント>

- SELECT 表現は通常の SQL 文と同様に記述しますが、抽出したデータを各変数に格納するために、INTO 句を最後に付加しなくてはなりません。
- 以下の例は、システムテーブルの中からテーブルだけを取りだして一覧として返すストアードプロシージャです。単に一覧を返すだけでなく SQL 文でも簡単にできますので、連番を付けてみました。

```

SET TERM ^;
CREATE PROCEDURE SP_ALL_TABLES
RETURNS(
    TAB_NO INTEGER,
    tab_name CHAR(31)
    CHARACTER SET UNICODE_FSS)
AS
BEGIN
    TAB_NO = 1;
    FOR SELECT rdb$relation_name
    FROM rdb$relations
    WHERE rdb$flags = 1
    AND rdb$view_source IS NULL
    INTO :tab_name DO
    BEGIN
        SUSPEND;
        TAB_NO = TAB_NO + 1;
    END
END ^
SET TERM ; ^

```

- 次の例はサンプルデータベース/**Employee.gdb**に含まれているものです。一つ目の **For Select** 文の結果を利用して、二つ目の **For Select** 文を実行しています。**FROM**句ではテーブルではなく、ストアードプロシージャ **show_langs** をしています。面白いのは、セパレーターとして「=」を戻り値にして、**SUSPEND** を実行しているところです。

```

SET TERM ^;
CREATE PROCEDURE ALL_LANGS
RETURNS
( CODE VARCHAR(5),
  GRADE VARCHAR(5),
  COUNTRY VARCHAR(15),
  LANG VARCHAR(15))
AS
BEGIN
    FOR SELECT job_code, job_grade, job_country
    FROM job
    INTO :code, :grade, :country
    DO
    BEGIN
        FOR SELECT languages FROM show_langs
        (:code, :grade, :country)
        INTO :lang DO
            SUSPEND;
            /* Put nice separators between rows */
            code = '=====';
            grade = '=====';
            country = '=====';
            lang = '=====';
            SUSPEND;
        END
    END^
SET TERM ; ^

```

IF ... THEN ... ELSE

おなじみの、**IF ... THEN ... ELSE** 構文です。プログラミングを少しでもかじったことのある人なら知らないはずがないでしょう。**IF** に続く条件が真(**TRUE**)の場合、**THEN** に続くブロック/ステートメントが実行されます。**ELSE** は任意指定ですが、条件が偽(**FALSE**)の場合に、ブロック/ステートメントが実行されます。

IF ... THEN ... ELSE 構文はトリガーとストアードプロシージャの両方で使用可能です。

```

IF (条件式) THEN <ブロック | ステートメント>
ELSE <ブロック | ステートメント>

```

- PSQL** と通常のプログラミング言語との最大の相違点は、**NULL** を含んだ演算が **NULL** を返すというところにあります。**NULL** は **TRUE** でも **FALSE** でもない **UNKNOWN** (不定) として取り扱われますので、注意してください。以下の例では、**NULL** による演算を回避するために **IF ... THEN ... ELSE** 構文を利用しています。ただし、**IBOConsole** のストアードプロシージャ・エディターでパラメータを指定して実行すると、**NULL** が文字列の「**NULL**」に

置き換えられてしまうので、効果がわかりません。

```
SET TERM ^ ;
CREATE PROCEDURE SP_TEST_NULL
(
    FIRST_NAME CHAR(10) CHARACTER SET ASCII,
    LAST_NAME CHAR(10) CHARACTER SET ASCII
)
RETURNS
(
    FULL_NAME CHAR(21) CHARACTER SET ASCII
)
AS
BEGIN
    IF (LAST_NAME IS NULL) THEN
        FULL_NAME = FIRST_NAME;
    ELSE
        BEGIN
            IF (FIRST_NAME IS NULL) THEN
                FULL_NAME = LAST_NAME;
            ELSE
                FULL_NAME =
                    FIRST_NAME || ' ' || LAST_NAME;
        END
    END
SUSPEND;
END^
SET TERM ; ^
```

ISQL での実行結果は以下のようになります。

```
SQL> SELECT *
      FROM SP_TEST_NULL(' TSUTOMU' , NULL);
```

```
FULL_NAME
=====
Tsutomu
```

```
SQL> SELECT *
      FROM SP_TEST_NULL(NULL, ' HAYASHI');
```

```
FULL_NAME
=====
Hayashi
```

```
SQL> SELECT *
      FROM SP_TEST_NULL(' TSUTOMU' , ' HAYASHI');
```

```
FULL_NAME
=====
Tsutomu   Hayashi
```

EXECUTE PROCEDURE

EXECUTE PROCEDURE 文は、トリガーとストアードプロシージャの中で、ストアードプロシージャを実行するために使用します。

```
EXECUTE PROCEDURE プロシージャ名
[:パラメータ [, :パラメータ...]]
[RETURNING_VALUES :パラメータ [, :パラメータ...]]
```

- ストアードプロシージャが入力パラメータを必要とする場合、省略することはできません。例えば、2つの入力パラメータを要求するプロシージャの実行に際して、2つ目のパラメータに NULL を指定する場合も、「NULL」と記述する必要があります。
- EXECUTE PROCEDURE 文中で指定するパラメータには変数と定数のどちらでも使用できます。また、パラメータの前には必ず「:」を付けなくてはなりません。
ただし、NEW と OLD のコンテキスト変数を使用する場合には、「:」は必要ありません。
- トリガー/ストアードプロシージャの中から、ストアードプロシージャを呼び出すことは、「ネストした呼び出し」と呼ばれています。ストアードプロシージャの中で、自分自身をよびだすことは「再帰的呼び出し」と呼ばれています。

ネスト/再帰呼び出しは、1000 レベルまでに制限されています。この制限により、終了条件が指定されていない場合に、プロシージャが無限ループに陥ることが防がれています。

サーバーのメモリーやスタックの条件により、ネストのレベル数が 1000 以下になる場合があります。

- 次の例では再帰的呼び出しによって、最大公約数を求めています。

```
SET TERM ^ ;
CREATE PROCEDURE SP_GCD
( M INTEGER, N INTEGER )
RETURNS
( GCD INTEGER )
AS
DECLARE VARIABLE MOD_MN INTEGER ;
BEGIN
IF (N = 0) THEN GCD = M;
ELSE
BEGIN
MOD_MN = MOD(M N);
EXECUTE PROCEDURE SP_GCD :N, :MOD_MN
RETURNING_VALUES GCD;
END
SUSPEND;
END^
SET TERM ; ^
```

ここでは、`ib_udf.dll` に含まれる `MOD()` というユーザー定義関数を利用しています。`ib_udf.dll` は Windows 版 Firebird のインストールキットに含まれる、UDF ライブラリです。Linux 版では、`ib_udf` です。

実行すると以下のようになります。

```
SQL> EXECUTE PROCEDURE SP_GCD 3, 9;
      GCD
=====
      3
```

- `MOD()` 関数をストアードで実装すると、以下のようになります。

```
SET TERM ^ ;
CREATE PROCEDURE SP_MOD
( M INTEGER, N INTEGER )
RETURNS
( RET INTEGER )
AS
BEGIN
RET = M - (M / N)*N;
SUSPEND;
END^
SET TERM ; ^
```

ISQL での実行例は以下のようになります。

```
SQL> EXECUTE PROCEDURE SP_MOD 5, 3;
      RET
=====
      2
```

WHILE ... DO

指定した条件式が TRUE(真)である間、DO の後で指定したブロック又はステートメントを実行します。トリガーとストアードプロシージャの両方で使用可能です。

WHILE (条件式) **DO** <ブロック|ステートメント>

- 条件式の判定は、ループに入る前に行われ、以降のループ毎に行われます。
- 以下の例では、`WHILE ... DO` ループを利用して、階乗を求めています。

```
SET TERM ^ ;
```

```

CREATE PROCEDURE SP_FACTORIAL
( M INTEGER )
RETURNS
( RET INTEGER )
AS
BEGIN
    RET = M;
    WHILE (M > 1) DO
    BEGIN
        M = M - 1;
        RET = RET * M;
    END
    SUSPEND;
END^
SET TERM ; ^

```

ISQL での実行例は以下のようになります。

```

SQL> EXECUTE PROCEDURE SP_FACTORIAL 5;
      RET
=====
      120

```

EXIT

EXIT 文は、最後の END 文にジャンプし、プロシージャーの実行を終了します。トリガーでは使用できません。

EXIT;

- EXIT 文は選択型プロシージャーと実行型プロシージャーの両方で使用できます。最後の END 文にジャンプした後の動作は、選択型プロシージャーと実行型プロシージャーで異なります。
- 選択型プロシージャーの場合は、制御を呼び出し元に戻して、SQLCODE を 100 に設定し、これ以上返す行がないことを指示します。
- 実行型プロシージャーの場合は、制御を呼び出し元に戻して、出力パラメータがある場合は、値を返します。

SUSPEND

SUSPEND 文は、選択型プロシージャーで使用し、SELECT 文に対して戻り値を返して、プロシージャーの実行を一時中断します。この動作により、選択型プロシージャーはテーブルと同様に振る舞うことができます

SUSPEND;

- SUSPEND 文を実行する前に、すべての出力パラメータに値を割り当てておいてください。未割り当ての出力パラメータには、NULL が返されます。
- 実行型プロシージャーで、SUSPEND 文を使用した場合、最後の END にジャンプして、実行を終了します。この使用法は推奨されません。
- 以下の例では、指定した整数までの階乗をそれぞれ一覧で表示するために、SUSPEND 文を使用しています。

```

SET TERM ^;
CREATE PROCEDURE SP_PARAM
( N INTEGER )
RETURNS
( RET1 INTEGER, RET2 INTEGER )
AS
BEGIN
    RET1 = 0;
    RET2 = 1;
    WHILE (RET1 < N) DO
    BEGIN
        RET1 = RET1 + 1;
        RET2 = RET1 * RET2;
        SUSPEND;
    END
END^

```

```
SET TERM ;^
```

- 上記のプロシージャーを、SELECT 文と EXECUTE PROCEDURE 文で使用した場合、それぞれ以下のような結果を返します。

```
SQL> SELECT * FROM SP_PARAM(5);
```

RET1	RET2
1	1
2	2
3	6
4	24
5	120

```
SQL> EXECUTE PROCEDURE SP_PARAM(5);
```

RET1	RET2
1	1

※一見してわかるとおり、EXECUTE PROCEDURE コマンドの実行結果は期待されるものではありません。

POST_EVENT

データベースに接続中のクライアントに対して、イベントの発生を通知します。各クライアントは **Firebird** に対して通知されるイベントを事前に登録しておく必要があります。

POST_EVENT 'イベント名' | 列 | 変数

- イベントとして処理されるのは文字型のデータに限られます。イベント名は最大 **15** 文字までになります。列/変数もそれぞれ文字型のデータを格納していなければなりません。
- イベントが **Post** されると、**Firebird** サーバーのイベントマネージャーは登録されている、イベントパラメータバッファに従って、同期/非同期で待機しているクライアントアプリケーションにイベントの発生を通知します。
- 次の例では、トリガーによってデータが更新されたことをクライアントアプリケーションに対して通知しています。

```
SET TERM ; !!
CREATE TRIGGER FOR EMPLOYEE AFTER INSERT
AS
BEGIN
    POST_EVENT 'INS_EMPLOYEE';
END!!
SET TERM !! ;
```

- クライアントアプリケーションは、応答するイベントを事前に登録しておかなくてはならないので、列や変数を使ったイベントのポストは利用が難しいのではないかと思います。考えられる利用法としては、都道府県名のようなあらかじめ分類されているデータについて、それぞれに対応するイベントを発生させるなどという利用法が考えられるかもしれません。また、特定のユーザーに対するイベントを発生させるために、ユーザー名が格納されている列を利用して以下のようなイベントを発生させることもあり得ます。その場合、クライアントアプリケーションは使用中のユーザー名に従って、イベントの登録を事前にしておく必要があります。

```
SET TERM ; !!
CREATE TRIGGER FOR EMPLOYEE
POST_EVENT 'ALERT_' || CAST(NEW EMP_NO AS CHAR(8));
```

EXCEPTION

ユーザーによって定義された例外を、指定して発行します。トリガーとストアプロシージャーの両方で使用できます。

EXCEPTION 例外名;

- 例外を発行したトリガー/ストアプロシージャーを終了し、トリガー/ストアプロシージャーによって行われた操作を（直接的又は間接的に）取り消します。
- 呼び出し元のアプリケーションにはエラーメッセージが返されます。**isql** の場合、エラーメッセージは画面に表示されます。
- 以下の例で、EXCEPTION 文の動作を見てみます。事前に例外 **EXP_TEST** を作成しておきます。

```
CREATE EXCEPTION EXP_TEST
```

```
'This is a test exception';

SET TERM ^ ;
CREATE PROCEDURE SP_EXCEPTION
( N INTEGER )
RETURNS
( RET INTEGER )
AS
BEGIN
    RET = N;
    SUSPEND;
    EXCEPTION EXP_TEST;
END^
SET TERM ; ^
```

```
SQL> SELECT * FROM SP_EXCEPTION(1);
      RET
=====
      1
```

Statement failed, SQLCODE = -836

```
exception 1
-This is a test exception
```

```
SQL> EXECUTE PROCEDURE SP_EXCEPTION(1);
      RET
=====
      1
```

※EXECUTE PROCEDURE コマンドの場合、SUSPEND によって最後の END までジャンプするので、例外が発生していないことに注意してください。

トリガーで使用できるコンテキスト変数

Firebird のトリガーでは、テーブル/ビューに対する操作に伴って挿入/更新/削除の前後で操作の対象となるデータを参照したり更新したりすることができます。Oracle の PL/SQL では「:new」と「:old」にあたるものが NEW コンテキスト変数と OLD コンテキスト変数です。

一見してわかるとおり、この二つはそれぞれテーブル/ビューの操作に伴って変更されるデータの変更前/変更後の値を示しています。以下に各 DML 文とトリガーの設定位置による各コンテキスト変数の使用可否を示します。

DML	位置	BEFORE		AFTER	
		OLD	NEW	OLD	NEW
INSERT		×	○	×	△
UPDATE		○	○	○	△
DELETE		○	×	○	×

※○：使用可能、×：エラー、△：参照/変更が可能だが、変更についてはデータベースに反映されない。

NEW コンテキスト変数

INSERT 及び UPDATE 操作に対して、挿入/更新が行われた後の値を示します。トリガーの中でだけ使用可能です。

NEW 列名

- NEW コンテキスト変数は、挿入前にジェネレーターを利用して連番をセットする目的でよく利用されます。挿入/更新の際に新しいデータをロギングする目的などでよく使われます。また、挿入及び更新に伴って計算されたデータをテーブルに格納する場合などにも使用できます。こうした手法は、参照が多数有り計算項目や VIEW の使用がボトルネックになるようなシステムでは有効です。
- 以下の例は、サンプルデータベースでテーブル EMPLOYEE に設定されているトリガーです。更新後に起動し、更新前後の給与を比較して、変更があれば履歴テーブルに情報を格納します。

```
SET TERM !! ;
CREATE TRIGGER SAVE_SALARY_CHANGE
FOR EMPLOYEE AFTER UPDATE
AS
BEGIN
    IF (OLD.SALARY <> NEW.SALARY) THEN
```

```

INSERT INTO SALARY_HISTORY
(EMP_NO, CHANGE_DATE, UPDATER_ID,
OLD_SALARY, PERCENT_CHANGE)
VALUES
(OLD.EMP_NO, 'NOW', USER,
OLD.SALARY,
(NEW.SALARY - OLD.SALARY) * 100
/ OLD.SALARY);
END !!
SET TERM ;

```

- SQL 分の中で使用する場合でも、コンテキスト変数には「:」をつけません。
- 次の例では、更新前にジェネレーターから連番を設定しています。

```

SET TERM ; !!
CREATE TRIGGER FOR EMPLOYEE BEFORE INSERT
AS
BEGIN
NEW_EMP_NO = GEN_ID(EMP_NO_GEN, 1);
END
SET TERM !! ;

```

OLD コンテキスト変数

UPDATE 及び DELETE 操作に対して、更新/削除が行われる前の値を示します。トリガーの中でだけ使用可能です。

OLD. 列名

- OLD コンテキスト変数は、更新/削除前の情報を履歴テーブルに格納する目的などで使用されます。NEW コンテキスト変数の項で例を示しました。
- DELETE に伴う使用例を以下に示します。

```

SET TERM !!
CREATE TRIGGER FOR EMPLOYEE AFTER DELETE
AS
BEGIN
INSERT INTO EMPLOYEE_RESIGNED
(EMP_NO, FIRST_NAME ,
LAST_NAME, PHONE_EXT,
HIRE_DATE, DEPT_NO,
JOB_CODE, JOB_GRADE,
JOB_COUNTRY, SALARY,
FULL_NAME, DELETE_DATE)
VALUES
(OLD.EMP_NO, OLD.FIRST_NAME,
OLD.LAST_NAME, OLD.PHONE_EXT,
OLD.HIRE_DATE, OLD.DEPT_NO,
OLD.JOB_CODE, OLD.JOB_GRADE,
OLD.JOB_COUNTRY, OLD.SALARY,
OLD.FULL_NAME, 'NOW');
END!!
SET TERM !! ;

```

WHEN ... DO ...

指定したエラーが起きた場合に、DO に続くブロック又は PSQL 文を実行します。

```

WHEN {<エラー>[,<エラー> ... ] | ANY}
DO <ブロック | ステートメント>
<エラー> =
{EXCEPTION 例外名 |
SQLCODE 数値 |
GDSCODE エラーコード}

```

- WHEN 文を使用する場合は、BEGIN ... END ブロックの一番最後に配置します。SUSPEND 文があるときも、その後にくるようにすべきです。

- **WHEN** 文で処理できる例外は、以下のトリガー/プロシーチャーの **EXCEPTION** 文で発行されたものです。
 - ①現在実行しているプロシーチャー
 - ②ネストして実行されているプロシーチャー
 - ③現在実行しているプロシーチャーの動作によって 起動したトリガー
- **SQLCODE** によって報告された **SQL** エラー
- **Firebird** エラーコード

例外処理

プロシーチャーの内部で例外が発生した場合、**Firebird** は以下のような手順で例外を処理します。

- 例外が発生した **BEGIN ... END** ブロックの実行を終了して、そのブロックで実行された動作を取り消します。
 - 1 レベル前の **BEGIN ... END** ブロックに戻って、**WHEN** 文を探します。これを、**WHEN** 文が見つかるまで繰り返します。**WHEN** 文が見つからない場合は、プロシーチャーを終了し、全ての動作を取り消します。
 - **WHEN** 文が見つかった場合は、その後続く文/ブロックを実行します。
 - **WHEN** 文の後に続く文/ブロックへと、プログラムの制御を戻します。
- ※**WHEN** 文の中で発生した例外からはエラーメッセージは返されず、直ちにプロシーチャーが終了します。

SQL エラーの処理

SQLCODE 番号で指示されたエラーも処理することが可能です。**SQL** 文を実行した場合、必ず成功か失敗かを示す **SQLCODE** が返されますので、これを利用可能です。

SQLCODE は巻末の「**Firebird** エラーコードとエラーメッセージ一覧」をご覧ください。

Firebird エラーコードの処理

プロシーチャーでは、**Firebird** エラーコードを処理することも可能です。このエラーコードは、たとえばジェネレータが定義されていないことを示す、**isc_gennotdef** など、**isc_hogehoge** といった名称で定義されています。本書では、分量の関係から一覧の掲載を見送りましたが、何らかの方法で公開できるようにしたいと思います。

例外処理の例

それでは、例外処理の例を以下に示したいと思います。以下の例では、外部キー違反を検出して、例外処理を行います。

```
SET TERM !!;
CREATE PROCEDURE SP_FOREIGN(ID1 INTEGER, ID INTEGER)
  RETURNS (RET VARCHAR(12) CHARACTER SET ASCII)
AS
  DECLARE VARIABLE MN_ID1 INTEGER;
BEGIN
  SELECT MN(ID) FROM T_TEST INTO :MN_ID1;
  BEGIN
    INSERT INTO T_TEST2(ID, ID1) VALUES (:ID, :ID1);
    RET = 'INSERT OK';
    WHEN SQLCODE -530 DO
      BEGIN
        INSERT INTO T_TEST2(ID, ID1)
          VALUES (:ID, :MN_ID1);
        RET = 'INSERT ERROR';
      END
    END
  END
  SUSPEND;
END
END!!
SET TERM !!
```

さて、このプロシーチャーをコンパイルして実行してみます。事前に以下のテーブルを用意します。

```
CREATE TABLE T_TEST
(
  ID      INTEGER NOT NULL,
  NAME    CHAR(10) CHARACTER SET ASCII,
  CONSTRAINT PK_T_TEST PRIMARY KEY (ID)
);
CREATE TABLE T_TEST2
(
  ID      INTEGER NOT NULL,
```

```

ID1      INTEGER NOT NULL,
CONSTRAINT PK_T_TEST2 PRIMARY KEY (ID)
);
ALTER TABLE T_TEST2 ADD CONSTRAINT FK_T_TEST2_1 FOREIGN KEY (ID1) REFERENCES T_TEST (ID) ON UPDATE CASCADE ON DELETE NO ACTION;

```

このテーブルには、それぞれ以下のようなデータが入っています。

```

SQL> SELECT * FROM T_TEST;
      ID NAME
=====
      1 poteneko
      2 potesi
      3 maki-neko

```

```

SQL> SELECT * FROM T_TEST2;
      ID      ID1
=====
      1          1
      3          3
      2          2

```

では、ストアプロシージャを実行してみます。

```

SQL> SELECT * FROM SP_FOREIGN(1, 4);
RET
=====
INSERT OK

```

```

SQL> SELECT * FROM SP_FOREIGN(5, 5);
RET
=====
INSERT ERROR

```

一回目の実行は、ID1 が 1、つまり T_TEST の ID 項目が 1 であるため、エラーになりません。
 2 回目の実行は、ID が 5 なので、T_TEST に対応するデータが存在しないため、エラーとなります。
 この場合、MN_ID1 で指定される、T_TEST.ID の最小値がセットされるようになっています。結果を見てみましょう。

```

SQL> SELECT * FROM T_TEST2;
      ID      ID1
=====
      1          1
      3          3
      2          2
      4          1
      5          1 <--ここです。

```

こうして、エラーが回避されて、ID1 に 1 がセットされていることが確認できました。

ストアードプロシージャの利用例

さて、この章ではストアードプロシージャの利用法を探っていきたいと思います。最初の題材はクロス集計です。クロス集計については、Oracle では `DECODE()` 関数を利用して集計するという方法がありますが、Firebird にはそうした便利な関数はないため、クライアント側で処理するかストアードを使う事になります。いくつかの方法が考えられるので、順を追って考察していくことにします。

サンプルデータの作成

まず、集計の対象となるテーブルを用意します。また、ジェネレーターを用意して、トリガーで主キーを設定するようしておきます。

```
CREATE TABLE T_SALES
( ID INTEGER NOT NULL,
  PRODUCT_ID INTEGER,
  QTY INTEGER,
  UNIT_PRICE INTEGER,
  DISCOUNT INTEGER,
  SALES_DATE DATE,
  CONSTRAINT PK_T_SALES PRIMARY KEY (ID)
);

CREATE GENERATOR GEN_SALES_ID;

SET TERM ^ ;
CREATE TRIGGER TRIG_T_SALES_BI FOR T_SALES
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  NEW ID = GEN_ID(GEN_SALES_ID, 1);
END^
SET TERM ; ^
```

次にこのテーブルに、サンプルデータを追加しておきます。どうせですから、サンプルデータの追加もストアードを使ってやってみましょう。

UDF の作成

ただし、一つ問題があります。サンプルデータを作成するためには乱数が必要になってきますが、標準で付属してくる `IB_UDF.DLL` の `RAND()` 関数が毎回初期化を行うようになっていて、非常に短い時間に連続して呼び出した場合、同じ値を返してしまうという問題があります。そこで、今回は仕方がないので `TOM_RAND()` という関数を含む UDF も作成しました。Delphi のコードを以下に示します。

```
library TomnekoUDF;

uses
  SysUtils,
  Classes;

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

function Tom_Randmi ze: Integer; cdecl;
begin
  Randomize;
  Result := 1;
end;

exports
  Tom_Rand,
  Tom_Randmi ze;

end.
```

これをコンパイルすると `TomnekoUDF.dll` ができますので、<Firebird ルートフォルダ>\UDF にコピーしておきます。Delphi プロジェクトではないので、コマンドラインでコンパイルします。

```
>dcc32 TomnekoUDF.pas
Borland Delphi Version 14.0
Copyright (c) 1983,2002 Borland Software Corporation
TomnekoUDF.Pas(24)
25 lines, 0.07 seconds, 118092 bytes code, 4565 bytes data.
```

さらに、これをデータベースに追加する SQL スクリプトを作成します。

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

DECLARE EXTERNAL FUNCTION TOM_RAND
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT 'Tom_Rand'  MODULE_NAME 'TOMNEKOUDF';

/* function Tom_Randmi ze: integer;
*/

DECLARE EXTERNAL FUNCTION TOM_RANDMIZE
RETURNS INTEGER BY VALUE
ENTRY_POINT 'Tom_Randmi ze'  MODULE_NAME 'TOMNEKOUDF';
```

サンプルデータ追加用のストアプロシージャの作成

では、サンプルデータを生成してテーブルに追加するストアを作ってみます。データの生成基準は以下のようになりました。

```
PRODUCT_ID : 1~15 の整数
QTY : 5~100 の整数
UNIT_PRICE : PRODUCT_ID×1000
DISCOUNT : 0~1500 の整数
SALES_DATE : 2003/5/1~2003/5/31 の日付
```

作成したストアプロシージャは以下のようになります

```
SET TERM ^ ;
CREATE PROCEDURE SP_INSERT_SALES
( N INTEGER )
RETURNS
( RET CHAR(30) ASCII)
AS
DECLARE VARIABLE V_PID INTEGER;
DECLARE VARIABLE V_QTY INTEGER;
DECLARE VARIABLE V_UNT INTEGER;
DECLARE VARIABLE V_DCN INTEGER;
DECLARE VARIABLE V_DAY INTEGER;
DECLARE VARIABLE V_DATE TIMESTAMP;
BEGIN
IF (N < 0) THEN
BEGIN
RET = 'No data inserted';
SUSPEND;
EXIT;
END

WHILE (N > 0) DO
BEGIN
V_PID = TOM_RAND() * 14 + 1;
V_QTY = TOM_RAND() * 95 + 5;
V_UNT = V_PID * 1000;
V_DCN = FLOOR(TOM_RAND() * 21) * 100 - 500;
IF (V_DCN < 0) THEN V_DCN = 0;
V_DAY = TOM_RAND() * 30;
V_DATE = ADDDAY('2003/5/1', V_DAY);
INSERT INTO T_SALES (PRODUCT_ID, QTY, UNIT_PRICE, DISCOUNT, SALES_DATE)
VALUES (:V_PID, :V_QTY, :V_UNT, :V_DCN, :V_DATE);
N = N - 1;
END
```

```

RET = 'Insert succeeded';
SUSPEND;

WHEN ANY DO
BEGIN
    RET = 'An error occured';
    SUSPEND;
END
END^
SET TERM ;^

```

isql から実行する場合、実行型/選択型のどちらでも実行できて、同じ結果を返します。

```

SQL> execute procedure sp_insert_sales(10000);
RET

```

```

=====
Insert succeeded

```

```

SQL> delete from t_sales;
SQL> select * from sp_insert_sales(10000);
RET

```

```

=====
Insert succeeded

```

例外の追加

ストアプロシージャでパラメータチェックに利用する例外を追加しておきます。年と月が Firebird のデータ型で扱える範囲を超えた場合に例外を発生して、終了するようにするためです。

```

CREATE EXCEPTION EXP_INVALID_MONTH
'Mnth must be between 1 and 12';
CREATE EXCEPTION EXP_INVALID_YEAR
'Year must be between 1 and 9999';

```

クロス集計ストアプロシージャ(1)

さて、ではサンプルデータに対してクロス集計を行うストアプロシージャを作成していきます。まず、集計方法に関しては、商品 ID をタテに、日付をヨコに取って、ある年月における毎日の商品毎の売上数量を集計するものとします。

商品 ID/日付 1, 2, 3, ... 31

```

1      数量...
2      数量...
.      .
.      .
.      .
15     数量...

```

まずは、手続き的に全部展開してみましょう。入力パラメータと出力パラメータは後で示すストアプロシージャでも共通となっています。

```

SET TERM ^ ;
CREATE PROCEDURE SP_CROSS_SUM
( P_MONTH INTEGER,
  P_YEAR INTEGER )
RETURNS
( PID INTEGER,
  X1 INTEGER, X2 INTEGER, X3 INTEGER, X4 INTEGER,
  X5 INTEGER, X6 INTEGER, X7 INTEGER, X8 INTEGER,
  X9 INTEGER, X10 INTEGER, X11 INTEGER, X12 INTEGER,
  X13 INTEGER, X14 INTEGER, X15 INTEGER, X16 INTEGER,
  X17 INTEGER, X18 INTEGER, X19 INTEGER, X20 INTEGER,
  X21 INTEGER, X22 INTEGER, X23 INTEGER, X24 INTEGER,
  X25 INTEGER, X26 INTEGER, X27 INTEGER, X28 INTEGER,
  X29 INTEGER, X30 INTEGER, X31 INTEGER )
AS
BEGIN
    /* INPUT PARAMETER CHECK */
    IF ((P_MONTH < 1) OR (P_MONTH > 12)) THEN

```

```

EXCEPTION EXP_INVALID_MONTH;
IF ((P_YEAR < 1) OR (P_YEAR > 9999)) THEN
EXCEPTION EXP_INVALID_YEAR;

FOR SELECT DISTINCT S.PRODUCT_ID
FROM T_SALES S
WHERE EXTRACT(MONTH FROM S.SALES_DATE) = :P_MONTH
AND EXTRACT(YEAR FROM S.SALES_DATE) = :P_YEAR
ORDER BY 1
INTO :PID
DO
BEGIN
SELECT SUM(S.QTY) FROM T_SALES S
WHERE S.PRODUCT_ID = :PID
AND EXTRACT(DAY FROM S.SALES_DATE) = 1
AND EXTRACT(MONTH FROM S.SALES_DATE) = :P_MONTH
AND EXTRACT(YEAR FROM S.SALES_DATE) = :P_YEAR
INTO :X1;

SELECT SUM(S.QTY) FROM T_SALES S
WHERE S.PRODUCT_ID = :PID
AND EXTRACT(DAY FROM S.SALES_DATE) = 2
AND EXTRACT(MONTH FROM S.SALES_DATE) = :P_MONTH
AND EXTRACT(YEAR FROM S.SALES_DATE) = :P_YEAR
INTO :X2;

~中略~

SELECT SUM(S.QTY) FROM T_SALES S
WHERE S.PRODUCT_ID = :PID
AND EXTRACT(DAY FROM S.SALES_DATE) = 31
AND EXTRACT(MONTH FROM S.SALES_DATE) = :P_MONTH
AND EXTRACT(YEAR FROM S.SALES_DATE) = :P_YEAR
INTO :X31;
SUSPEND;
END
END^
SET TERM ; ^

```

一見してわかるとおり、年と月を指定した範囲に存在する **PRODUCT_ID** を利用して、順次 1 日の数量、2 日の数量・・・31 日の数量と集計していきます。SELECT 文の発行回数は **PRODUCT_ID** の数×31 となっています。

実行してみるとわかりますが、大変時間がかかります。約 44 秒かかりました。これでは意味がありません。

クロス集計ストアプロシージャー(2)

では、次に日付でグループ化してから集計するように変更してみましょう。SELECT 文で **EXTRACT (DAY FROM SALES_DATE)** を列にとって、**GROUP BY** 句でグループ化したいところですが、単に **EXTRACT()** を指定すると **GROUP BY** に指定することができません。これは、InterBase からの制約で、**GROUP BY** 句には内部関数を指定できないためです。

Firebird でも、内部関数を直接指定することはできませんが、UDF を使用した列を指定することは可能です。そのため、内部関数を使用した列も UDF でラッピングすることで **GROUP BY** 句に指定することが可能となります。これを **UDF ラッパー** といいます。

例を挙げると **bin_or()** という **ib_udf.dll** で定義されている UDF は、2 つの整数パラメータに対して、ビットレベルの **OR** 演算を行います。したがって、**bin_or(0, N)** という演算の結果はかならず N になります。あるいは、**round()** という **fb_udf.dll** で定義されている UDF は、実数パラメータに対して、四捨五入を行います。ですから、**round(integer N)** の結果は、かならず N になります。

この性質を利用すると、以下のような SQL が可能となります。

```

SELECT BIN_OR(0, EXTRACT(DAY FROM S.SALES_DATE)),
SUM(S.QTY)
FROM T_SALES S
WHERE EXTRACT(MONTH FROM S.SALES_DATE) = 5
AND EXTRACT(YEAR FROM S.SALES_DATE) = 2003
AND S.PRODUCT_ID = 1
GROUP BY BIN_OR(0, EXTRACT(DAY FROM S.SALES_DATE))

```

実行した結果は以下のようになります。

```

BIN_OR          SUM
=====

```

```

1          343
2          787
~ 中略 ~

31         144

```

このようにして得られた結果に対して、横へ展開をすればいいわけです。では、集計用のストアードプロシージャーを作成してみます。

```

SET TERM ^ ;
CREATE PROCEDURE SP_CROSS_SUM2
( P_MONTH INTEGER,
  P_YEAR INTEGER )
RETURNS
( PID INTEGER,
  X1 INTEGER, X2 INTEGER, X3 INTEGER, X4 INTEGER,
  X5 INTEGER, X6 INTEGER, X7 INTEGER, X8 INTEGER,
  X9 INTEGER, X10 INTEGER, X11 INTEGER, X12 INTEGER,
  X13 INTEGER, X14 INTEGER, X15 INTEGER, X16 INTEGER,
  X17 INTEGER, X18 INTEGER, X19 INTEGER, X20 INTEGER,
  X21 INTEGER, X22 INTEGER, X23 INTEGER, X24 INTEGER,
  X25 INTEGER, X26 INTEGER, X27 INTEGER, X28 INTEGER,
  X29 INTEGER, X30 INTEGER, X31 INTEGER )
AS
  DECLARE VARIABLE V_DAY INTEGER;
  DECLARE VARIABLE V_QTY INTEGER;
BEGIN
  /* INPUT PARAMETER CHECK */
  IF ((P_MONTH < 1) OR (P_MONTH > 12)) THEN
    EXCEPTION EXP_INVALID_MONTH;
  IF ((P_YEAR < 1) OR (P_YEAR > 9999)) THEN
    EXCEPTION EXP_INVALID_YEAR;

  FOR SELECT DISTINCT S.PRODUCT_ID
  FROM T_SALES S
  WHERE EXTRACT(MONTH FROM S.SALES_DATE) = :P_MONTH
    AND EXTRACT(YEAR FROM S.SALES_DATE) = :P_YEAR
  ORDER BY 1
  INTO :PID
  DO
  BEGIN
    FOR
    SELECT BIN_OR(0, EXTRACT(DAY FROM S.SALES_DATE))
      P_DAY, SUM(S.QTY) P_QTY
    FROM T_SALES S
    WHERE
      EXTRACT(MONTH FROM S.SALES_DATE) = :P_MONTH
      AND EXTRACT(YEAR FROM S.SALES_DATE) = :P_YEAR
      AND S.PRODUCT_ID = :PID
    GROUP BY
      BIN_OR(0, EXTRACT(DAY FROM S.SALES_DATE))
    ORDER BY 1
    INTO :V_DAY, :V_QTY
    DO
    BEGIN
      IF (V_DAY = 1) THEN X1 = V_QTY;
      IF (V_DAY = 2) THEN X2 = V_QTY;
      IF (V_DAY = 3) THEN X3 = V_QTY;
      IF (V_DAY = 4) THEN X4 = V_QTY;
      IF (V_DAY = 5) THEN X5 = V_QTY;
      IF (V_DAY = 6) THEN X6 = V_QTY;
      IF (V_DAY = 7) THEN X7 = V_QTY;
      IF (V_DAY = 8) THEN X8 = V_QTY;
      IF (V_DAY = 9) THEN X9 = V_QTY;
      IF (V_DAY = 10) THEN X10 = V_QTY;
      IF (V_DAY = 11) THEN X11 = V_QTY;
      IF (V_DAY = 12) THEN X12 = V_QTY;
      IF (V_DAY = 13) THEN X13 = V_QTY;

```

```

    IF (V_DAY = 14) THEN X14 = V_QTY;
    IF (V_DAY = 15) THEN X15 = V_QTY;
    IF (V_DAY = 16) THEN X16 = V_QTY;
    IF (V_DAY = 17) THEN X17 = V_QTY;
    IF (V_DAY = 18) THEN X18 = V_QTY;
    IF (V_DAY = 19) THEN X19 = V_QTY;
    IF (V_DAY = 20) THEN X20 = V_QTY;
    IF (V_DAY = 21) THEN X21 = V_QTY;
    IF (V_DAY = 22) THEN X22 = V_QTY;
    IF (V_DAY = 23) THEN X23 = V_QTY;
    IF (V_DAY = 24) THEN X24 = V_QTY;
    IF (V_DAY = 25) THEN X25 = V_QTY;
    IF (V_DAY = 26) THEN X26 = V_QTY;
    IF (V_DAY = 27) THEN X27 = V_QTY;
    IF (V_DAY = 28) THEN X28 = V_QTY;
    IF (V_DAY = 29) THEN X29 = V_QTY;
    IF (V_DAY = 30) THEN X30 = V_QTY;
    IF (V_DAY = 31) THEN X31 = V_QTY;
END
SUSPEND;
END
END^
SET TERM ; ^

```

今度は約 1.1 秒で終了しました。これならなんとか実用に耐える速度ではないでしょうか。

クロス集計ストアドプロシージャ(3)

さて、(2)の例では UDF ラッパーを用いて、内部関数を使用した列をグループ化しました。こうしたテクニックを使わないで集計を行う場合には、別の選択型ストアドプロシージャを作成して利用します。

以下に示すストアドプロシージャは、月・年・商品 ID を指定して呼び出すと、日付と数量を返してきます。この結果セットに対して集計を行えば、グループ化は普通に行うことができます。

```

CREATE PROCEDURE SP_SALES_MONTHLY_QTY
( P_MONTH INTEGER,
  P_YEAR INTEGER,
  P_PID INTEGER )
RETURNS
( P_DAY INTEGER,
  P_QTY INTEGER )
AS
BEGIN
  /* INPUT PARAMETER CHECK */
  IF ((P_MONTH < 1) OR (P_MONTH > 12)) THEN
    EXCEPTION EXP_INVALID_MONTH;
  IF ((P_YEAR < 1) OR (P_YEAR > 9999)) THEN
    EXCEPTION EXP_INVALID_YEAR;

  FOR
  SELECT EXTRACT(DAY FROM S.SALES_DATE), S.QTY
  FROM T_SALES S
  WHERE S.PRODUCT_ID = :P_PID
        AND EXTRACT(MONTH FROM S.SALES_DATE) = :P_MONTH
        AND EXTRACT(YEAR FROM S.SALES_DATE) = :P_YEAR
  INTO :P_DAY, :P_QTY
  DO SUSPEND;
END^
SET TERM ; ^

SET TERM ^ ;
CREATE PROCEDURE SP_CROSS_SUMB
( P_MONTH INTEGER,
  P_YEAR INTEGER )
RETURNS
( PID INTEGER,
  X1 INTEGER, X2 INTEGER, X3 INTEGER, X4 INTEGER,
  X5 INTEGER, X6 INTEGER, X7 INTEGER, X8 INTEGER,
  X9 INTEGER, X10 INTEGER, X11 INTEGER, X12 INTEGER,

```

```

X13 INTEGER, X14 INTEGER, X15 INTEGER, X16 INTEGER,
X17 INTEGER, X18 INTEGER, X19 INTEGER, X20 INTEGER,
X21 INTEGER, X22 INTEGER, X23 INTEGER, X24 INTEGER,
X25 INTEGER, X26 INTEGER, X27 INTEGER, X28 INTEGER,
X29 INTEGER, X30 INTEGER, X31 INTEGER )
AS
DECLARE VARIABLE V_DAY INTEGER;
DECLARE VARIABLE V_QTY INTEGER;
BEGIN
/* INPUT PARAMETER CHECK */
IF ((P_MONTH < 1) OR (P_MONTH > 12)) THEN
EXCEPTION EXP_INVALID_MONTH;
IF ((P_YEAR < 1) OR (P_YEAR > 9999)) THEN
EXCEPTION EXP_INVALID_YEAR;

FOR SELECT DISTINCT S.PRODUCT_ID
FROM T_SALES S
WHERE EXTRACT(MONTH FROM S.SALES_DATE) = :P_MONTH
AND EXTRACT(YEAR FROM S.SALES_DATE) = :P_YEAR
ORDER BY 1
INTO :PID
DO
BEGIN
FOR
SELECT S.P_DAY, SUM(S.P_QTY)
FROM
SP_SALES_MONTHLY_QTY(:P_MONTH, :P_YEAR, :PID) S
GROUP BY S.P_DAY
ORDER BY 1
INTO :V_DAY, :V_QTY
DO
BEGIN
IF (V_DAY = 1) THEN X1 = V_QTY;
IF (V_DAY = 2) THEN X2 = V_QTY;
IF (V_DAY = 3) THEN X3 = V_QTY;
IF (V_DAY = 4) THEN X4 = V_QTY;
IF (V_DAY = 5) THEN X5 = V_QTY;
IF (V_DAY = 6) THEN X6 = V_QTY;
IF (V_DAY = 7) THEN X7 = V_QTY;
IF (V_DAY = 8) THEN X8 = V_QTY;
IF (V_DAY = 9) THEN X9 = V_QTY;
IF (V_DAY = 10) THEN X10 = V_QTY;
IF (V_DAY = 11) THEN X11 = V_QTY;
IF (V_DAY = 12) THEN X12 = V_QTY;
IF (V_DAY = 13) THEN X13 = V_QTY;
IF (V_DAY = 14) THEN X14 = V_QTY;
IF (V_DAY = 15) THEN X15 = V_QTY;
IF (V_DAY = 16) THEN X16 = V_QTY;
IF (V_DAY = 17) THEN X17 = V_QTY;
IF (V_DAY = 18) THEN X18 = V_QTY;
IF (V_DAY = 19) THEN X19 = V_QTY;
IF (V_DAY = 20) THEN X20 = V_QTY;
IF (V_DAY = 21) THEN X21 = V_QTY;
IF (V_DAY = 22) THEN X22 = V_QTY;
IF (V_DAY = 23) THEN X23 = V_QTY;
IF (V_DAY = 24) THEN X24 = V_QTY;
IF (V_DAY = 25) THEN X25 = V_QTY;
IF (V_DAY = 26) THEN X26 = V_QTY;
IF (V_DAY = 27) THEN X27 = V_QTY;
IF (V_DAY = 28) THEN X28 = V_QTY;
IF (V_DAY = 29) THEN X29 = V_QTY;
IF (V_DAY = 30) THEN X30 = V_QTY;
IF (V_DAY = 31) THEN X31 = V_QTY;
END
SUSPEND;
END
END^
SET TERM ; ^

```

この例では実行に約 2 秒かかりました。UDF ラッパーのオーバーヘッドの方が、サブクエリー代替の選択型ストアドプロシージャーのオーバーヘッドよりも大きいようです。

クロス集計ストアドプロシージャー(4)

さて、クロス集計を行う場合、上述したような手続きのに展開した方法をとることもできますが、行列変換を利用してさらに高速化する方法があります。

まず、変換用のテーブルを用意します。

```
CREATE TABLE T_CROSSDAY
( DAY_ID  INTEGER NOT NULL,
  D1      SMALLINT,
  D2      SMALLINT,
  D3      SMALLINT,
  D4      SMALLINT,
  D5      SMALLINT,
  D6      SMALLINT,
  D7      SMALLINT,
  D8      SMALLINT,
  D9      SMALLINT,
  D10     SMALLINT,
  D11     SMALLINT,
  D12     SMALLINT,
  D13     SMALLINT,
  D14     SMALLINT,
  D15     SMALLINT,
  D16     SMALLINT,
  D17     SMALLINT,
  D18     SMALLINT,
  D19     SMALLINT,
  D20     SMALLINT,
  D21     SMALLINT,
  D22     SMALLINT,
  D23     SMALLINT,
  D24     SMALLINT,
  D25     SMALLINT,
  D26     SMALLINT,
  D27     SMALLINT,
  D28     SMALLINT,
  D29     SMALLINT,
  D30     SMALLINT,
  D31     SMALLINT,
  PRIMARY KEY (DAY_ID)
);
```

このテーブルに以下のようなマトリックスとなるデータを格納しておきます。

```
DAY_ID, D1, D2, D3, .... D29, D30, D31
1      , 1, 0, 0, .... 0, 0, 0
2      , 0, 1, 0, .... 0, 0, 0
3      , 0, 0, 1, .... 0, 0, 0
~ 中略 ~
31     , 0, 0, 0, .... 0, 0, 1
```

さて、ではこの変換テーブルを使ったストアドプロシージャーを作成してみます。

```
SET TERM ^ ;
CREATE PROCEDURE SP_CROSS_SUMM
( P_MONTH INTEGER,
  P_YEAR  INTEGER )
RETURNS
( PID INTEGER,
  X1 INTEGER, X2 INTEGER, X3 INTEGER, X4 INTEGER,
  X5 INTEGER, X6 INTEGER, X7 INTEGER, X8 INTEGER,
  X9 INTEGER, X10 INTEGER, X11 INTEGER, X12 INTEGER,
  X13 INTEGER, X14 INTEGER, X15 INTEGER, X16 INTEGER,
  X17 INTEGER, X18 INTEGER, X19 INTEGER, X20 INTEGER,
```

```

X21 INTEGER, X22 INTEGER, X23 INTEGER, X24 INTEGER,
X25 INTEGER, X26 INTEGER, X27 INTEGER, X28 INTEGER,
X29 INTEGER, X30 INTEGER, X31 INTEGER )

```

```
AS
```

```
BEGIN
```

```
/* INPUT PARAMETER CHECK */
```

```
IF ((P_MONTH < 1) OR (P_MONTH > 12)) THEN
```

```
EXCEPTION EXP_INVALID_MONTH;
```

```
IF ((P_YEAR < 1) OR (P_YEAR > 9999)) THEN
```

```
EXCEPTION EXP_INVALID_YEAR;
```

```
FOR
```

```
SELECT S.PRODUCT_ID,
```

```
SUM(S.QTY * C.D1) AS X1,
```

```
SUM(S.QTY * C.D2) AS X2,
```

```
SUM(S.QTY * C.D3) AS X3,
```

```
SUM(S.QTY * C.D4) AS X4,
```

```
SUM(S.QTY * C.D5) AS X5,
```

```
SUM(S.QTY * C.D6) AS X6,
```

```
SUM(S.QTY * C.D7) AS X7,
```

```
SUM(S.QTY * C.D8) AS X8,
```

```
SUM(S.QTY * C.D9) AS X9,
```

```
SUM(S.QTY * C.D10) AS X10,
```

```
SUM(S.QTY * C.D11) AS X11,
```

```
SUM(S.QTY * C.D12) AS X12,
```

```
SUM(S.QTY * C.D13) AS X13,
```

```
SUM(S.QTY * C.D14) AS X14,
```

```
SUM(S.QTY * C.D15) AS X15,
```

```
SUM(S.QTY * C.D16) AS X16,
```

```
SUM(S.QTY * C.D17) AS X17,
```

```
SUM(S.QTY * C.D18) AS X18,
```

```
SUM(S.QTY * C.D19) AS X19,
```

```
SUM(S.QTY * C.D20) AS X20,
```

```
SUM(S.QTY * C.D21) AS X21,
```

```
SUM(S.QTY * C.D22) AS X22,
```

```
SUM(S.QTY * C.D23) AS X23,
```

```
SUM(S.QTY * C.D24) AS X24,
```

```
SUM(S.QTY * C.D25) AS X25,
```

```
SUM(S.QTY * C.D26) AS X26,
```

```
SUM(S.QTY * C.D27) AS X27,
```

```
SUM(S.QTY * C.D28) AS X28,
```

```
SUM(S.QTY * C.D29) AS X29,
```

```
SUM(S.QTY * C.D30) AS X30,
```

```
SUM(S.QTY * C.D31) AS X31
```

```
FROM T_SALES S, T_CROSSDAY C
```

```
WHERE EXTRACT(DAY FROM S.SALES_DATE) = C.DAY_ID
```

```
AND EXTRACT(MONTH FROM S.SALES_DATE) = :P_MONTH
```

```
AND EXTRACT(YEAR FROM S.SALES_DATE) = :P_YEAR
```

```
GROUP BY S.PRODUCT_ID
```

```
INTO
```

```
:PID, :X1, :X2, :X3, :X4, :X5, :X6, :X7, :X8,
```

```
:X9, :X10, :X11, :X12, :X13, :X14, :X15, :X16,
```

```
:X17, :X18, :X19, :X20, :X21, :X22, :X23, :X24,
```

```
:X25, :X26, :X27, :X28, :X29, :X30, :X31
```

```
DO
```

```
SUSPEND;
```

```
END^
```

```
SET TERM ; ^
```

さて、実行してみると 0.8 秒で終了しました。あまり差はありませんが、この方法が一番速いようです。また、変換テーブルを用意するのが面倒だという気もしますので、変換テーブルを返す選択型ストアプロシージャを作成してみましょう。

```
SET TERM ^;
```

```
CREATE PROCEDURE SP_CROSS_31
```

```
RETURNS
```

```
( DAY_ID INTEGER,
```

```
D1 SMALLINT, D2 SMALLINT, D3 SMALLINT,
```

```
D4 SMALLINT, D5 SMALLINT, D6 SMALLINT,
```

```

D7 SMALLINT, D8 SMALLINT, D9 SMALLINT,
D10 SMALLINT, D11 SMALLINT, D12 SMALLINT,
D13 SMALLINT, D14 SMALLINT, D15 SMALLINT,
D16 SMALLINT, D17 SMALLINT, D18 SMALLINT,
D19 SMALLINT, D20 SMALLINT, D21 SMALLINT,
D22 SMALLINT, D23 SMALLINT, D24 SMALLINT,
D25 SMALLINT, D26 SMALLINT, D27 SMALLINT,
D28 SMALLINT, D29 SMALLINT, D30 SMALLINT,
D31 SMALLINT)
AS
BEGIN
  DAY_ID = 0;
  WHILE (DAY_ID < 31) DO
  BEGIN
    DAY_ID = DAY_ID + 1;
    IF (DAY_ID = 1) THEN D1 = 1; ELSE D1 = 0;
    IF (DAY_ID = 2) THEN D2 = 1; ELSE D2 = 0;
    IF (DAY_ID = 3) THEN D3 = 1; ELSE D3 = 0;
    IF (DAY_ID = 4) THEN D4 = 1; ELSE D4 = 0;
    IF (DAY_ID = 5) THEN D5 = 1; ELSE D5 = 0;
    IF (DAY_ID = 6) THEN D6 = 1; ELSE D6 = 0;
    IF (DAY_ID = 7) THEN D7 = 1; ELSE D7 = 0;
    IF (DAY_ID = 8) THEN D8 = 1; ELSE D8 = 0;
    IF (DAY_ID = 9) THEN D9 = 1; ELSE D9 = 0;
    IF (DAY_ID = 10) THEN D10 = 1; ELSE D10 = 0;
    IF (DAY_ID = 11) THEN D11 = 1; ELSE D11 = 0;
    IF (DAY_ID = 12) THEN D12 = 1; ELSE D12 = 0;
    IF (DAY_ID = 13) THEN D13 = 1; ELSE D13 = 0;
    IF (DAY_ID = 14) THEN D14 = 1; ELSE D14 = 0;
    IF (DAY_ID = 15) THEN D15 = 1; ELSE D15 = 0;
    IF (DAY_ID = 16) THEN D16 = 1; ELSE D16 = 0;
    IF (DAY_ID = 17) THEN D17 = 1; ELSE D17 = 0;
    IF (DAY_ID = 18) THEN D18 = 1; ELSE D18 = 0;
    IF (DAY_ID = 19) THEN D19 = 1; ELSE D19 = 0;
    IF (DAY_ID = 20) THEN D20 = 1; ELSE D20 = 0;
    IF (DAY_ID = 21) THEN D21 = 1; ELSE D21 = 0;
    IF (DAY_ID = 22) THEN D22 = 1; ELSE D22 = 0;
    IF (DAY_ID = 23) THEN D23 = 1; ELSE D23 = 0;
    IF (DAY_ID = 24) THEN D24 = 1; ELSE D24 = 0;
    IF (DAY_ID = 25) THEN D25 = 1; ELSE D25 = 0;
    IF (DAY_ID = 26) THEN D26 = 1; ELSE D26 = 0;
    IF (DAY_ID = 27) THEN D27 = 1; ELSE D27 = 0;
    IF (DAY_ID = 28) THEN D28 = 1; ELSE D28 = 0;
    IF (DAY_ID = 29) THEN D29 = 1; ELSE D29 = 0;
    IF (DAY_ID = 30) THEN D30 = 1; ELSE D30 = 0;
    IF (DAY_ID = 31) THEN D31 = 1; ELSE D31 = 0;
  SUSPEND;
  END
END^
SET TERM ; ^

```

さて、こうして用意した SP_CROSS_31 を T_CROSSDAY と置き換えて実行してみたところ、1.7 秒で終了しました。やはり、多少のオーバーヘッドはあるようですが、許容範囲かと思えます。

ただし、これらの方法の難点は、テーブルを用意するにせよ、ストアードプロシージャで対応するにせよ、列数を動的に変更することができないという点です。この点については、クライアント側で対応するしかないでしょう。

指定した一連番号を返すストアードプロシージャ

さて、今度は趣向を変えて連続した一連のユニークキーが削除などによって、とびとびになっている場合に、その欠番を探すために利用できるストアードプロシージャを作成してみます。

以下に、開始番号と終了番号を指定すると、連番を返すストアードプロシージャを作成します。

```

SET TERM ; ^
CREATE PROCEDURE SP_SEQUENCE
(INT_FIRST INTEGER, INT_LAST INTEGER)
RETURNS
(SEQ_NO INTEGER)
AS

```

```

DECLARE VARIABLE CNT INTEGER;
BEGIN
  CNT = INT_FIRST;
  WHILE (CNT <= INT_LAST) DO
  BEGIN
    SEQ_NO = CNT;
    SUSPEND;
    CNT = CNT + 1;
  END
END^
SET TERM ^ ;

```

実行すると以下のような結果を返します。

```

SQL> SELECT * FROM SP_SEQUENCE(1, 10);
      SEQ_NO
=====
          1
          2
          3
          4
          5
          6
          7
          8
          9
         10

```

指定した範囲の欠番を返す SQL は以下ようになります。

```

SELECT SQ.SEQ_NO, S.ID
FROM SP_SEQUENCE(1, 100) SQ
LEFT JOIN T_SALES S
ON SQ.SEQ_NO = S.ID
WHERE S.ID IS NULL;

```

この場合、SP_SEQUENCE の終了番号を T_SALES の行数にしたいのですが、COUNT(S.ID)などと指定することはできないので、まず、行数を取得してから、SELECT するようなストアドプロシージャを作成します。

```

SET TERM ^ ;
CREATE PROCEDURE SP_SALES_ID_DROPPED
RETURNS
( ID INTEGER )
AS
  DECLARE VARIABLE ROW_COUNT INTEGER;
BEGIN
  SELECT COUNT(*) FROM T_SALES INTO :ROW_COUNT;
  FOR
  SELECT SQ.SEQ_NO
  FROM SP_SEQUENCE(1, :ROW_COUNT) SQ
  LEFT JOIN T_SALES S
  ON SQ.SEQ_NO = S.ID
  WHERE S.ID IS NULL
  INTO :ID
  DO SUSPEND;
END^
SET TERM ; ^

```

テストのためにいくつかの行を削除してから、実行してみます。

```

SQL> DELETE FROM T_SALES WHERE ID = 3;
SQL> DELETE FROM T_SALES WHERE ID = 7;
SQL> SELECT * FROM SP_SALES_ID_DROPPED;
      ID
=====
          3
          7

```

Elapsed time= 0.21 sec

トリガーの使用例

この章ではトリガーの使用例について取り上げたいと思います。トリガーは、テーブルの削除/変更/挿入の前後に、それらの動作をきっかけとして起動されます。したがって、**CHECK** 制約ではチェックできないようなデータのチェックに使用したり、あるいはテーブルの更新に伴うロギングに使用したり、主キーのオートインクリメントに使用したりします。

ジェネレータを使用したオートインクリメント

まず、もっともよく使用されるであろうオートインクリメント・トリガーの実装を示します。テーブル **T_TEST** の主キーが **ID** 列であった場合を想定すると以下ようになります。

```
CREATE GENERATOR GEN_T_TEST_ID;

SET TERM ^;
CREATE TRIGGER TRG_T_TEST_BI FOR T_TEST
ACTIVE AFTER INSERT POSITION 0
AS
BEGIN
    NEW.ID = GEN_ID(GEN_TEST_ID, 1);
END ^
SET TERM ;^
```

トリガーによるロギング

今度は、トリガーによるロギングの例を考えてみます。良くあげられる例としては、給与情報の更新に伴って前給と新給与をログテーブルに保管するというモノです。まずテーブルを用意します。

```
CREATE TABLE T_SARARY
(ID INTEGER NOT NULL PRIMARY KEY,
NAME VARCHAR(10) NOT NULL,
SARARY INTEGER NOT NULL);

CREATE TABLE T_SARARY_HISTORY
(ID INTEGER NOT NULL PRIMARY KEY,
NAME VARCHAR(10) NOT NULL,
OLD_SARARY INTEGER NOT NULL,
NEW_SARARY INTEGER NOT NULL,
CHANGE_RATE DOUBLE PRECISION NOT NULL,
CHANGE_TIMESTAMP TIMESTAMP DEFAULT 'NOW' NOT NULL);
```

各テーブルの主キーには、オートナンバーを設定しておきます。

```
CREATE GENERATOR GEN_T_SARARY_ID;
CREATE GENERATOR GEN_T_SARARY_HISTORY_ID;

SET TERM ^;
CREATE TRIGGER TRG_T_SARARY_BI FOR T_SARARY
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
    NEW.ID = GEN_ID(GEN_T_SARARY_ID, 1);
END^
SET TERM ;^

SET TERM ^;
CREATE TRIGGER TRG_T_SARARY_HISTORY_BI
FOR T_SARARY_HISTORY
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
    NEW.ID = GEN_ID(GEN_T_SARARY_HISTORY_ID, 1);
END^
SET TERM ;^

INSERT INTO T_SARARY
(NAME, SARARY)
VALUES
('TOMNEKO', 200000);
INSERT INTO T_SARARY
(NAME, SARARY)
```

```
VALUES
('MAKINEKO', 250000);
INSERT INTO T_SARARY
(NAME, SARARY)
VALUES
('PORENEKO', 15000);
```

さて、では履歴テーブルへのロギングを行うトリガーを書いてみましょう。

```
SET TERM ^;
CREATE TRIGGER TRG_T_SARARY_AF FOR T_SARARY
ACTIVE AFTER UPDATE POSITION 0
AS
  DECLARE VARIABLE NEW_SARARY DOUBLE PRECISION;
  DECLARA VARIABLE OLD_SARARY DOUBLE PRECISION;
BEGIN
  NEW_SARARY = NEW.SARARY;
  OLD_SARARY = OLD.SARARY;
  INSERT INTO T_SARARY_HISTORY
  (NAME, OLD_SARARY, NEW_SARARY, CHANGE_RATE)
  VALUES
  (OLD.NAME, OLD.SARARY, NEW.SARARY,
  (:NEW_SARARY - :OLD_SARARY) / :OLD_SARARY);
END^
SET TERM ;^
```

ここで、注意すべき点としては、NEW.SARARY と OLD.SARARY を直接扱って演算を行った場合、両者のデータ型が INTEGER であるため、演算の結果も INTEGER として扱われてしまうということです。そのため、NEW_SARARY と OLD_SARARY という DOUBLE PRECISION 型のローカル変数を用意して、演算を行うようにしてあります。国内通貨を扱う場合には、小数点以下の数値はひとまずありませんが、演算精度が必要な場合には NUMERIC や DECIMAL などの高精度数値データ型を利用して、NUMERIC(18,2) などとすべきでしょう。

さて、それではデータを更新してみましょう。

```
SQL> SELECT * FROM T_SARARY;
```

ID	NAME	SARARY
4	TOMNEKO	200000
5	MAKINEKO	250000
6	PORENEKO	15000

```
SQL> UPDATE T_SARARY SET SARARY=210000 WHERE NAME = 'TOMNEKO';
SQL> SELECT * FROM T_SARARY;
```

ID	NAME	SARARY
4	TOMNEKO	210000
5	MAKINEKO	250000
6	PORENEKO	15000

```
SQL> SELECT * FROM T_SARARY_HISTORY;
```

ID	NAME	OLD_SARARY	NEW_SARARY	CHANGE_RATE	CHANGE_TIMESTAMP
4	TOMNEKO	200000	210000	0.050000000	2003-05-27 20:53:08.0000

ビュートリガーによる更新制御

Firebird は単一のテーブルの一部をなすような単純なビューに対する直接の更新をサポートしています。アクセス制御などを目的として、このようなビューを利用する場合、テーブルと同様に扱うことができます。

一方、複数のテーブルからなるより複雑なビューを使用する場合、当然にも直接の更新を行うことはできません。その場合は、ビューに対してトリガーを作成し、削除/更新/挿入を制御することで、こうしたビューもテーブルと同様に扱うことが可能となります。まず、以下のようなテーブルを 2 つ用意します。

```
CREATE TABLE TABLE1
( ID      INTEGER NOT NULL PRIMARY KEY,
  NAME    VARCHAR(10) CHARACTER SET ASCII);
```

```
CREATE TABLE TABLE2
( ID      INTEGER NOT NULL PRIMARY KEY,
  T1_ID   INTEGER NOT NULL
  REFERENCES TABLE1 (ID) ON UPDATE CASCADE,
  PHONE   VARCHAR(15) CHARACTER SET ASCII);
```

```
INSERT INTO TABLE1 VALUES(1, 'TOMNEKO');
INSERT INTO TABLE1 VALUES(2, 'MAKINEKO');
INSERT INTO TABLE1 VALUES(3, 'POTENNEKO');
```

```
INSERT INTO TABLE2 VALUES(1, 1, '090-1111-1111');
INSERT INTO TABLE2 VALUES(2, 1, '03-2222-2222');
INSERT INTO TABLE2 VALUES(3, 2, '090-3333-3333');
INSERT INTO TABLE2 VALUES(4, 2, '03-4444-4444');
INSERT INTO TABLE2 VALUES(5, 3, '090-5555-5555');
```

次に、この2つのテーブルを基にしたビューを作成します。

```
CREATE VIEW V_LIST
( ID1,
  ID2,
  NAME,
  PHONE) AS
SELECT T1.ID, T2.ID, T1.NAME, T2.PHONE
FROM TABLE1 T1, TABLE2 T2
WHERE T1.ID = T2.T1_ID;
```

さて、このビューに対する操作例を以下に示します。isql.exeでの実行例です。

```
SQL> SELECT * FROM V_LIST;
```

ID1	ID2	NAME	PHONE
1	1	TOMNEKO	090-1111-1111
1	2	TOMNEKO	03-2222-2222
2	3	MAKINEKO	090-3333-3333
2	4	MAKINEKO	03-4444-4444
3	5	POTENNEKO	090-5555-5555

```
SQL> UPDATE V_LIST SET PHONE = '090-1234-1234'
CON> WHERE ID1 = 1 AND ID2 = 1;
STATEMENT FAILED, SQLCODE = -150
CANNOT UPDATE READ-ONLY VIEW V_LIST
```

当然 Update することが出来ません。そこで、このビューに以下のトリガーを設定します。

```
SET TERM ^;
CREATE TRIGGER TRG_V_LIST_BU FOR V_LIST
ACTIVE BEFORE UPDATE POSITION 0
AS
BEGIN
  UPDATE TABLE1 SET NAME = NEW.NAME
  WHERE ID = OLD.ID1;
  UPDATE TABLE2 SET PHONE = NEW.PHONE
  WHERE ID = OLD.ID2;
END^
SET TERM ; ^
```

```
SQL> UPDATE V_LIST SET PHONE = '090-1234-1234'
CON> WHERE ID2 = 1 AND ID1 = 1;
SQL> SELECT * FROM V_LIST;
```

ID1	ID2	NAME	PHONE
1	1	TOMNEKO	090-1234-1234
1	2	TOMNEKO	03-2222-2222
2	3	MAKINEKO	090-3333-3333

```
2  4 MAKINEKO  03-4444-4444
3  5 POTENeko  090-5555-5555
```

ここで、例えば電話番号の更新は許可するが、名前の変更は認めないという場合、以下のようにすることで更新を制御することが出来ます。

```
CREATE EXCEPTION EXP_CANT_CHANGE_NAME
'YOU CAN NOT CHANGE USER NAME';
```

```
SET TERM ^;
ALTER TRIGGER TRG_V_LIST_BU
ACTIVE BEFORE UPDATE POSITION 0
AS
BEGIN
  IF (NEW.NAME <> OLD.NAME) THEN
    EXCEPTION EXP_CANT_CHANGE_NAME;
  ELSE
    UPDATE TABLE2 SET PHONE = NEW.PHONE
      WHERE ID = OLD.ID2;
END^
SET TERM ;^
```

```
SQL> UPDATE V_LIST SET NAME = 'TOMCAT' WHERE ID1 = 1;
Statement failed, SQLCODE = -836
exception 1
-you can not change name
SQL> update v_list set phone = '090-4321-4321' where id2 = 1;
SQL> select * from v_list;
```

ID1	ID2	NAME	PHONE
1	1	TOMNEKO	090-4321-4321
1	2	TOMNEKO	03-2222-2222
2	3	MAKINEKO	090-3333-3333
2	4	MAKINEKO	03-4444-4444
3	5	POTENeko	090-5555-5555

次に挿入に対する制御を考えてみます。行が挿入される場合、すでに存在するユーザーに対して、電話番号を追加することだけを許可するには以下のようにします。

```
SET TERM ^;
CREATE TRIGGER TRG_V_LIST_BI FOR V_LIST
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  INSERT INTO TABLE2
    VALUES(NEW.ID2, NEW.ID1, NEW.PHONE);
END^
SET TERM ;^
```

```
SQL> INSERT INTO V_LIST(ID1, ID2, PHONE) VALUES (1, 6, '090-1111-111');
SQL> SELECT * FROM V_LIST;
```

ID1	ID2	NAME	PHONE
1	1	TOMNEKO	090-4321-4321
1	2	TOMNEKO	03-2222-2222
2	3	MAKINEKO	090-3333-3333
2	4	MAKINEKO	03-4444-4444
3	5	POTENeko	090-5555-5555
1	6	TOMNEKO	090-1111-111

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 0.3333333333333333e0 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。

文字データ型

データ型	サイズ	範囲/有効桁数	説明
CHAR(n)	n 字	1～32,767 バイト キャラクタセットの文字サイズにより、32KB に納まる文字数。	固定長の CHAR 型またはテキスト文字列型。 CHARACTER キーワードも使用可。
VARCHAR(n)	n 文字	1 ～ 32,765 バイト。 キャラクタセットの文字サイズにより、32KB に納まる文字数。	可変長の CHAR 型 (テキスト文字列型)。 VARCHAR のかわりに、CHAR VARYING または、CHARACTER VARYING も使用できる。

Firebird では、上記の CHAR 型と VARCHAR 型の他に、NCHAR 型と NVARCHAR 型の 4 つの文字データ型をサポートしています。NCHAR 型・NVARCHAR 型は、ISO8859_1 キャラクタセットを使用する文字列型なので、本質的には固定長文字データ型と可変長文字データ型の 2 つだけをサポートしているといえます。

列のデータ型を指定する時に、CHARACTER SET オプションでキャラクタセットを指定することが出来ます。また、各キャラクタセットで定義されているコレーションオーダー（照合順序）を指定することが出来ます。

文字データ型で使用されるバイト数は、各キャラクタセットで必要とされるバイト数によって決まります。SJIS_0208 のような 2 バイトのキャラクタセットであれば、1 文字に 2 バイトが使用され、したがって最大文字数は 16,383 文字となります。これ以上の文字数を必要とする場合は、列を分けるか BLOB を使用してください。

- 固定長文字データ型では、指定された文字数に満たないデータについて Firebird は空白を付け加えて文字数を満たします。指定された文字数を越えたデータは切り捨てられます。デフォルトの文字数は 1 なので、CHAR は CHAR(1) と同じです。データの格納にあたっては、末尾の空白は圧縮されるので、CHAR 型が必ずしもディスクスペースを無駄にするわけではありません。
- 可変長文字データ型ではデフォルト値がないので、文字数 n は必ず指定しなくてはなりません。可変長文字データ型ではよりディスクスペースを節約できるので、検索が早くなりますが、挿入が遅くなる可能性が有ります。末尾に空白を付加したくない場合には、可変長文字データ型を使用してください。

日付時刻型

データ型	サイズ	範囲/有効桁数	説明
DATE	32 ビット	西暦 100 年 1 月 1 日～32768 年 2 月 29 日	年月日の日付情報のみを表す。
TIME	32 ビット	0:00 AM ～ 23:59.9999 PM	一日の午前零時からの時刻を 1/1000 秒単位で表す。
TIMESTAMP	64 ビット	西暦 100 年 1 月 1 日～32768 年 2 月 29 日	DATE と TIME の情報を組み合わせたデータ型。

Firebird では、DATE・TIME・TIMESTAMP の 3 つの日付時刻データ型をサポートしています。

- DATE は、日付を 32 ビットのデータとして格納します。西暦 100 年 1 月 1 日から 32768 年 2 月 29 日の範囲を有効な日付としてサポートします。
- TIME は、時刻を 32 ビットのデータとして格納します。00:00 AM から 23:59.9999 PM の範囲を有効な日付として認識します。最小単位は 1/1000 秒となります。
- TIMESTAMP は、2 つの 32 ビットのデータとして格納され、DATE と TIME を組み合わせたものとなっています。

BLOB データ型

データ型	サイズ	範囲/有効桁数	説明
BLOB	可変長		グラフィック、文字、音声データなどのサイズが動的に決まるデータ型。 内容はサブタイプによって決定される。

Firebird は動的にサイズの変更が可能な BLOB (Binary Large Object) データ型をサポートしています。BLOB データ型には、グラフィック、サウンド、動画、文字列などあらゆるデータを格納することが出来ます。また、BLOB フィルタを使用することによって、データの出力に伴って形式の変換を行うことなども可能です。