ワンデイ・スペシャル・トレーニング

ジャン＝ピエール・リブロウ（ Jean-Pierre Ribreau）

4D Developer Conference 2016

本文内容

はじめに

4Dは，着実に進化し続けているツールです。新しいコマンドや追加された機能は，是非とも活用したいものですが，反面，システムの複雑化が進めば，厄介なバグが混入し，貴重なプログラミング時間をその退治に奪われてしまうかもしれません。これは確かに悩ましい問題です。

4Dは，いわゆる『オブジェクト指向プログラミング言語（OOP）』ではありません。しかし，フォームには**オブジェクト**を配置することができますし，（JSONタイプの）**オブジェクト**を使用することもできます。近年のバージョンでは，独立した汎用的なLEGO™のような部品を組み合わせることにより，プログラミング手法に**オブジェクト志向**の方法論を取り入れることができるようになりました。こうしたものを上手に活用すれば，開発に充てる時間を節約し，検証に必要な時間を最小限に抑え，再利用の度合いと効率を高めることができます。

進歩のペースにしっかり付いてゆくために，コードのリユーズ（再利用），そしてフォームオブジェクトの弱い（緩い）カップリングというテクニックを活用することができます。このセミナーでは，はじめにデータベース定義に依存しないフォームオブジェクトの作り方とコードの書き方を習得します。次に，パラメーターや機能を記述するために，フォームオブジェクトの名前を利用する方法を学びます。各オブジェクトの特性と必要を見極め，1個のパラメーターを介して情報を渡すためには，4Dのイントロスペクション（内鑑）コマンドを使用します。このテクニックが目指すところは，フォームや4Dコードを変更することなく，テーブルやフィールドの追加と変更にすぐ対応できるようにすることです。このコンセプトは，リストボックス・サブフォーム・オブジェクトの動的な定義と複製など，さまざまな場所で応用することができます。他にも，便利な技や役立つ情報を随所で紹介してゆきます。

モダンなアプリケーションを設計するには

適切なインデックスタイプの選定

4Dは，いろいろなタイプのインデックスをサポートしています。

- B-ツリー
- クラスター
- 複合フィールド
- キーワード

これらは，基本的に下記いずれかの系統（ファミリー）に属するものです。

- B-ツリー系
- クラスター系

B-ツリー

インデックスの対象データは，**スカラー型**とそれ以外に分類されます。4Dの場合，スカラー型とは，固定サイズのフィールドに1度にひとつだけ保存できる値の単位のことを指します。具体的には，バイト・整数・倍長整数・64ビット整数・実数・ブール・日付型のことです。スカラー型には，通常，B-ツリータイプのインデックスを設定します。Bツリータイプのインデックスは，内部的にツリーのようなレイアウトのページ群になっています。1枚のページには，最大で255個の値が収められており，以下の構造をしています。

最初の255個までは，値を管理するために1枚のインデックスページが使用されます。256個目の値が追加されると，128個の値を持った第1ページと第2段階のページにインデックスが分割されます。第2階層のページが埋まるたびに，第2段階のページが追加され，こうしてツリーのバランスが均等に保たれます。こうしたことは，16,384個目の値まで続けられます。その後，第3段階のページが追加され，2,097,152個目の値まで管理できるようなツリー構造になります。第3段階のページがすべて埋まると，第4段階のページが作られ，こうして20億，つまりテーブルに保存できるレコード数の倍以上の**キー**が管理できるツリー構造が出来上がります。

B-ツリーインデックスの仕組みは次のように説明することができます。

仮に，値Xを探しているとしましょう。4Dは，第1インデックステーブルの中でこの値を探します。値がみつかった場合，対応するレコード番号がセレクションに追加されます。みつからなかった場合，つまり，該当する位置でみつかったのがレコード番号ではなく，ページ番号だった場合，そのページにアクセスして同じことを繰り返します。ページは4段階まで作られるため，レコード数が10億件でも**4回以内のディスクアクセス**でたどり着くことができます。

非スカラー型，つまり文字列・テキスト・複合フィールドの場合も，基本的な論理は同じですが，ページ毎のキー（またはエントリー）の数が128個に限定されている点が違います。それぞれのキーは，値とレコード番号に加え，**次の値までのオフセット**を情報として持っています。レコードサイズが一定ではないためです。なお，テキスト型については，最適化を図るため，冒頭の1024文字だけがインデックス管理の対象となっています。

埋まったページは分割され，インデックスのバランス，つまり階層の深さができるだけ均等に保たれるようになっています。

クラスター

クラスターインデックスは，B-ツリーの構造に基づいており，ほとんど仕組みが同じですが，それぞれのキーに対応する1個のレコード番号を保存する代わりに，以下に述べるような4バイトの値を使用する点が異なっています。

- キーに対応するレコード数が1件であればレコード番号
- キーに対応するレコード数が2件以上であればクラスター番号

クラスターは，レコード番号のリストであり，下記のいずれかの形を取ります。

- レコードの数が少ない（基本的には，テーブルに保存されているレコード数の1/32よりも少ない）場合，レコード番号（各4バイト）の配列。このリストの構造は，セレクションと同じです。
- それ以外の場合，テーブルのレコード1件に対して1ビットが割り当てられたビットテーブル。対象レコードに対応するビットは1，残りは0に値がセットされます。このリストの構造は，セットと同じです。

テーブルには10億件までレコードが保存できるので，クラスターは10億ビット，つまり約119MBに達する可能性があります。10億を掛ければ，なんでもおおきな数になるのでしょうけど・・・

そのようなわけで，おおきなクラスターは複数のページに分割され，すべてが同じ値であるページについては，圧縮して表現するといった最適化が図られています。

複合フィールド

異なるフィールド値を連結したものが1個のキーになる点が違っています。インデックスの構造は，B-Treeあるいはクラスター B-ツリーであり，1ページ毎のキー数は128個です。このインデックスを使用するかどうかは，4Dが自動的に判断します。複合インデックスは，セレクションを絞り込む点で通常のインデックスよりも優れています。たとえば，苗字="山田" かつ 名前="太郎" みたいなクエリの場合，本来であれば，"山田" のインデックス検索で作成されたセレクションと，"太郎" のインデックス検索で作成されたセレクションの論理演算を実行する必要がありますが，苗字+名前という複合インデックスが利用できれば，1回のクエリでセレクションが作成できるからです。しかし，複合インデックスのフィールド順序を入れ替えると，別の意味になってしまう点に留意してください。

キーワード

文字列やテキストから抽出された単語のリスト，あるいはピクチャに埋め込まれたメタデータに基づき，インデックスが作成されます。構造的には，クラスターB-ツリーインデックス（ページ毎128キー）であり，各キーは，1個の単語であり，関連づけられたクラスターには，フィールドの値にその単語が含まれているレコードのリストが保存されています。単語の定義は，ICU (International Components for Unicode) のルールに基づいており，単純にスペース記号で分割されるわけではありません。

注記: 日本語版は，MeCabライブラリv0.996/jumandic 7.0-20130310辞書に基づくキーワード抽出ができます。「データベース言語」を日本語に設定し，「非文字・非数字のみをキーワード区切り文字とする」を無効に設定してください。

「%」は，「単語を含む」という意味のテキスト比較演算子です。単純に「含む」という意味ではない点に留意してください。通常の文字列比較とは違い，テキストに含まれる「単語」そのものが評価の対象なので，たとえば単語の一部を構成する文字1個，あるいは複数の単語で構成されるフレーズをこの演算子で探しても，目的のレコードはみつかりません。「キーワード区切り文字」に挟まれた文字列が「単語」ですが，記号類は区切り文字，と決まっているわけではありません。たとえば，"today's" のような文字列に含まれるアポストロフィ記号は区切り文字ではなく，単語の一部です。ピリオドやカンマは，普通の文脈では区切り文字ですが，数値の桁区切りや小数点として用いられているときは単語の一部となります。しかし，通貨記号は単語の一部とならずに無視されます。

キーワードインデックスが設定されたフィールドに対してDISTINCT VALUESを使用すると，カレントセレクションに含まれる単語のリストが返されます。

内部的には，フィールドが変更されるたびにGET TEXT KEYWORDS (*) と同等の処理が実行され，返された単語のリストに基づいて対応するクラスターが更新されています。

同じフィールドに対して複数のインデックスを設定することもでき，その場合，4Dは検索を実行するのに最適と思われるインデックスを自動的に選択します。とはいえ，インデックスの管理には，コストがかかっているので注意が必要です・・・

憶えておきたい基本ルールを述べておきたいと思います。

- 良い気分になるのは使用している時だけ，普段はいつも最悪だ!

    an index is always painful, except when one needs it....

善し悪しは，状況に依存するということです。

全文検索: 最適化のコツ

4Dのキーワードインデックスは良くできていると思いますが，所詮は全自動システムです。アプリケーションの用途に応じ，独自のシステムを加えて改善できることでしょう。

たとえば，Googleのような「気の利いた」検索を実装するために，次のようなことができるかもしれません。インデックスが作成される前に，データを加工しておくのがポイントです。

1. 検索で使用されそうなワードだけに単語リストを絞り込む

    管理用のインデックスフィールドを用意する

    必要であればICU非区切り文字を置換する

    余計な単語をリストから除外する

2. 同意語・類義語・対義語などを追加する

3. ルールの例外は区切り文字を置換する: "vitamin-c" など

4. さまざまなスペリングに対応する

    語尾変化・合成語は語幹だけに縮約する

    単語を綴りではなくSoundex（発音に基づく表記）で管理する

クラスター: 過去と現在

クラスターは，ビットテーブル，つまりセットに似ています。ビットの位置は，レコード番号に対応しており，それぞれがアドレステーブルのエントリー1個と関連づけられています。昔のバージョンでは，クエリを高速化するために，セットを保存してハッシュテーブルを自作することがありました。4Dのクラスターは，内部的にビットテーブル（セット）とレコード番号のリスト（セレクション）を使い分けており，セットだけのクラスターよりも効率的です。

テキスト・BLOB・ピクチャを保存する場所の設定

年々，コンピューターの処理速度は速くなっていますが，ユーザーの感覚についていえば，機械と同じペースでスピードアップしているわけではありません。結果として，アプリケーションの速度を徹底的に意識して最適化することは，昔ほど重要なことではないように思えます。とはいえ，最適化がまったく必要ないわけではありません。たとえば，Webサーバーは何千件ものリクエストを瞬時にさばくことが求められますし，テーブルに数百万件のレコードが登録されていることは珍しくありません。フィールドにマルチメディアのデータが保存されていれば，ずっとおおきなディスク容量とメモリが必要です。ですから，やはり速度には気を遣わなければなりません。ミリ秒をナノ秒に改善することは，一見，無意味なこと（したがって時間の無駄）に思えるかもしれませんが，100万回，その処理を繰り返せば，ナノ秒はミリ秒，ミリ秒は20分に膨れ上がります。前者はピンとこないとしても，さすがに後者は無視できないのではないでしょうか・・・

では，いつ，最適化に取り組むべきでしょうか。プログラミングの第1原則がその答えです。

- 一概には言えないものだ

    it all depends....

データアクセスがアプリケーションのパフォーマンスに深刻な影響を与える主要な理由のひとつに，シーケンシャルアクセスの多発が挙げられます。4Dは，データファイルが空であれば，レコード番号1に続けてレコード番号2，といった風にレコードを保存してゆきます。シーケンシャルアクセスがもっとも快速に実行されるのは，この順番が維持されている場合ですから，後述するように，データの断片化を回避することは重要です。しかし，そんなことを気にしなくても良い場合もあることでしょう。昔のバージョンでは，データの断片化を回避するためのテクニックが知られていました。

- レコードサイズを均一にするための「エアーバッグ」

- レコードの一部（BLOB，ピクチャ，ときには長大なテキストも）を管理するための「ファットデータテーブル」

現バージョンの4Dでは，ストラクチャの設定でおおきなオブジェクトの保存場所を変更することができます。とはいえ，完璧な設定などというものはありません。どこか別の場所に保存したところで，レコードを参照すれば，大抵の場合，そのおおきなオブジェクトもロードされることになるからです。設定できる保存場所は以下のとおりです。

- レコードと一緒: ちいさなサイズに最適
- データファイルの中: ある程度，データファイルの断片化を抑えることができる
- データファイルの外: 別ファイルとして管理されるため，関係性が失われるリスクもある

加えて以下の選択肢も挙げることができます。

- 別テーブルで管理: 必要なときにだけロードする
- 別データファイル: エクスターナルデータベースなど

内部ストレージの最大サイズは，便利なオプションですが，所詮は自動オプションであり，データファイルの断片化をきっちり管理するという点ではいささか不十分です。

v12以降，ピクチャフィールドにもインデックスが設定できるようになりました。その場合，ピクチャに埋め込まれたキーワードに基づくクエリが最適化されます。メタデータは，SET PICTURE METADATAおよびGET PICTURE METADATAコマンドでも読み書きができます。

いずれにしても，可変長データを保存する場所と方法は，さまざまな要素を考慮した上で決定しなければなりません。

- アクセスはシーケンシャルか
- 使用するフィールドはまちまちか
- 可変長フィールドよ固定長フィールドの割合はどうか
- レコードを頻繁にロードする必要があるか
- クライアント/サーバーか

- レコード総数が多いか
- キャッシュに余裕があるか
- プロセス数は多いか
- ユーザーが待てないと感じる限界はどの程度なのか

オブジェクト型フィールド

近年，「スキーマレス」データベースという言葉を耳にす機会が増えました。有名なのはMongoDBでしょう。MongoDBは，JSONに似た形式のデータストレージです。あるいは，ドキュメント指向データベース，という言い方がされることもあります。データベースに保存されるドキュメントは，いろいろなタイプのフィールドを有しており，フィールドの組み合わせはドキュメント毎に違っています。強調しておきたい点ですが，スキーマが存在しない，というのは間違いです。レコードを訂正する必要が生じた場合，あるいはデータを移行しなければならない場合，スキーマが存在しなければ非常に困ります。「スキーマレス」データベースが「スキーマフル」データベースよりも扱いが簡単だ，ということはありません。

MongoDBの世界では「（既存の）オブジェクトが問題を引き起こすことはまずない」と良く言われます。しかし，ほんとうにそうでしょうか。「失敗する可能性のあるものは，失敗する（マーフィー）」の法則を思い出してください。

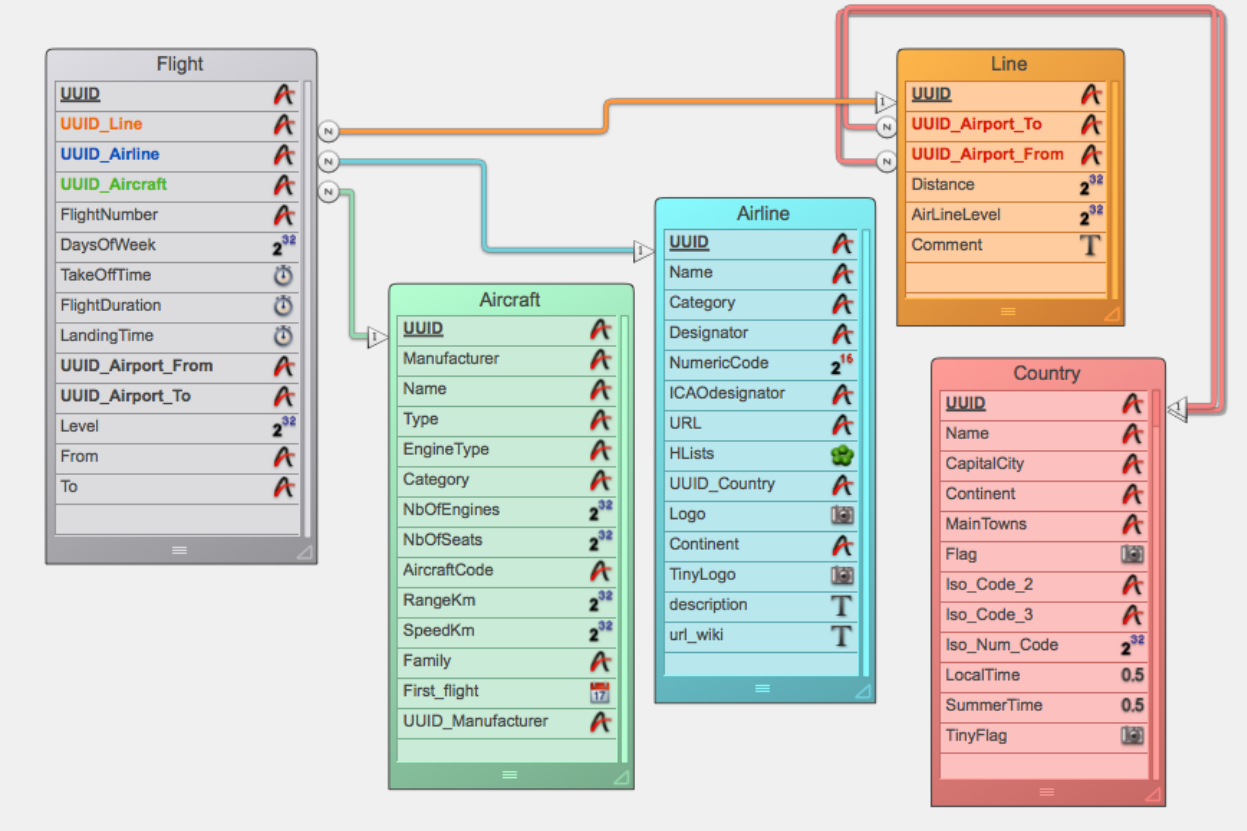
実際のところ，エンドユーザーがしばしば必要としているのは，ストラクチャを撤廃することではなく，ニーズに合わせてストラクチャを柔軟に変更できること，です。たとえば，顧客テーブルにペットの名前と誕生日を追加して欲しいとか，自動車のメーカーと年数も追加して欲しいとか，そういった要望があることでしょう。もしかしたら，そのような情報をBLOBやテキストの備考フィールドに突っ込んでいたかもしれません。しかし，それでは，ほんとうの意味でデータベースを改修したことにはなりません。データベースのフィールドである以上，SCRUD，つまり，**S**earch, **C**reate, **R**ead, **U**pdate, **D**elete，すべての操作がきちんとサポートされているべきです。

そこでオブジェクト型フィールドの出番です。4D v15以降で利用できるようになった新しいフィールド型を活用すれば，スキーマレスとスキーマフルの中間，いわば**スキーマフレキシブル**の概念をアプリケーションに持ち込むことができます。

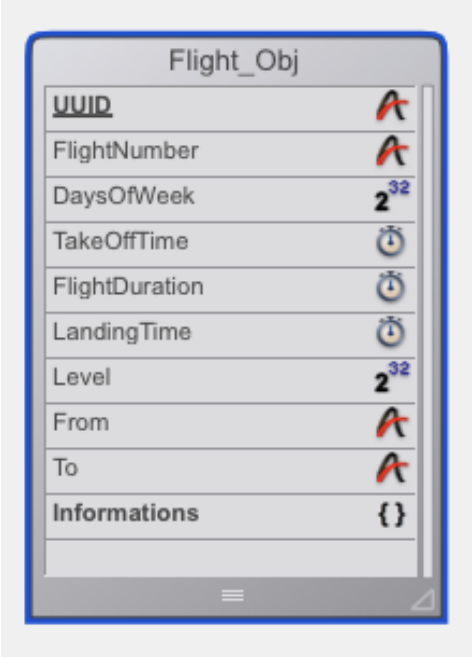
4D v15のオブジェクト型フィールドは，SCRUDの定義をしっかり満たしていますが，それはプログラミング・レベルでの話であり，QUERY BY ATTRIBUTE (v15), DISTINCT ATTRIBUTE PATHS (v16), DISTINCT ATTRIBUTE VALUES (v16) といったコマンドの使用が前提です。エンドユーザーにオブジェクト型フィールドをそのまま提供することはしないでください。相手はJSONが流暢に話せるわけではないのですから・・・

オブジェクト型フィールドで何ができるのか，誇張された，極端な例を考えてみましょう。

このサンプルは，4Dの古典的なリレーショナル・データベースです。



Informationsという名前のオブジェクト型フィールドを追加し， Airline, Aircraft, Line, Countryのデータをそこに移動すれば，テーブル数を1個に減らすことができます。



QueryOnObject フォームには，下記のようなフォームメソッドが記述されています。

```
If (Form event=On Load)
  ALL RECORDS([Flight_Obj])
  ARRAY TEXT(arr_Attributes;0)
  ARRAY TEXT(arr_Values;0)
End if
vTitleText:=String(Records in selection([Flight_Obj])
                                        ;"|LongInt")+" Records"
vNbSeats:=Sum([Flight_Obj]Informations;"aircraft.nbSeats")
vMinSeats:=Min([Flight_Obj]Informations;"aircraft.nbSeats")
vMaxSeats:=Max([Flight_Obj]Informations;"aircraft.nbSeats")
```

おなじみの統計関数，Sum, Min, Max, Averageは，シンタックスが拡張され（v16），"aircraft.nbSeats" のようなオブジェクトの**パス**を文字列で渡せるようになりました。

"Get it" ボタンのメソッドも非常にシンプルです。存在するパスのリストが表示されます。「可能な」パスのリストではないことに注目してください。

```
ALL RECORDS([Flight_Obj])
ARRAY TEXT(arr_Attributes;0)
```

```
DISTINCT ATTRIBUTE PATHS([Flight_Obj]Informations;arr_Attributes)
```

配列に表示されたパスをクリックしたときに実行されるのは，次のようなコードです。

```
ARRAY TEXT(arr_Values;0)
If (arr_Attributes{ar_Attributes}#"")
  DISTINCT ATTRIBUTE VALUES([Flight_Obj]Informations;
                  arr_Attributes{arr_Attributes};arr_Values)
End if
```

配列に表示された値をクリックしたときに実行されるのは，次のようなコードです。

```
If ((arr_Values>0) & (arr_Values<=Size of array(arr_Values)))
  QUERY BY ATTRIBUTE([Flight_Obj];[Flight_Obj]Informations;

arr_Attributes{arr_Attributes};=;arr_Values{arr_Values})
End if
```

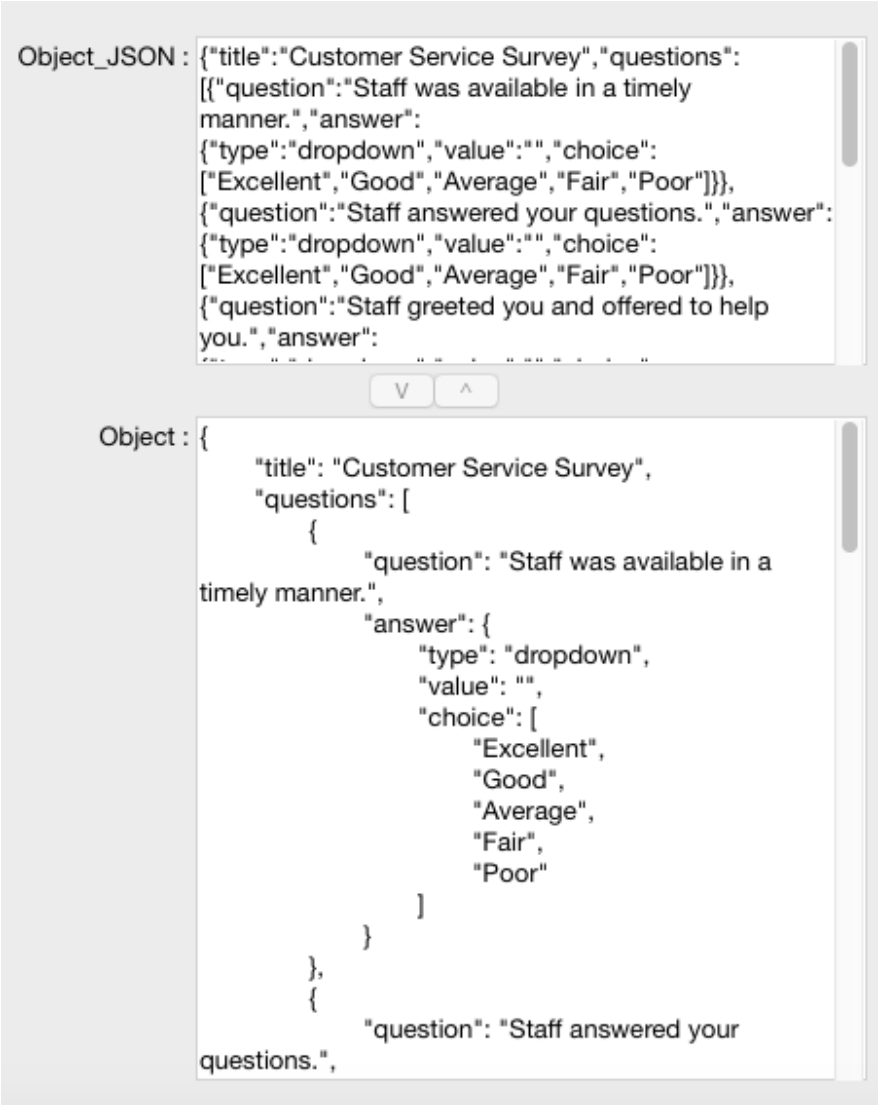
たったこれだけのコーディングで，従来の4Dデータベースと同じようなクエリがオブジェクト型フィールドによる Key-Value Store でも実現できました。

オブジェクト型フィールドの表示と入力

スキーマフレキシブルには欠かせないオブジェクト型フォールドですが，これを表示するためには，ユーザーが容易に理解でき，安全に操作できるように配慮されたインタフェースを用意しなければなりません。

サンプルデータベース "4DObjectsJPR" より， Test_Objects テーブルの入力フォームを開いてみてください。オブジェクト型フィールドのデータは，そのままJSONテキストとして表示されています。

一方は，改行コードやインデントが随所に挿入されており，JSONの構造がわかりやすくなっています。しかし，どちらの画面もエンドユーザーに易しいとは言い難いものです。



そのようなわけで，サンプルには，オブジェクトフィールドの内容を表示したり，編集したりするための画面が他にもいくつか用意されています。

これから紹介するのは，どんなアプリケーションにも簡単に移植できる，汎用的な例題です。ご自身のデータベース・ストラクチャにいくらかの柔軟性を追加した後，この例題をそのまま使用することができます。例題 4DOOP.4dbase を開き，名前が "OBJ_" から始まるプロジェクトメソッド，それから下記のフォームを探してください。

- DisplayObject_TXT
- DisplayObject_HL
- DisplayObject_LB
- OBJ_Edit_Element
- OBJ_ButtonBar

1. DisplayObject_TXT:

このフォームには，"_OBJ_TEXT_" という名称のスタイル付きテキスト入力エリアとボタンが配置されており，ボタンには，下記のようなスクリプトが記述されています。

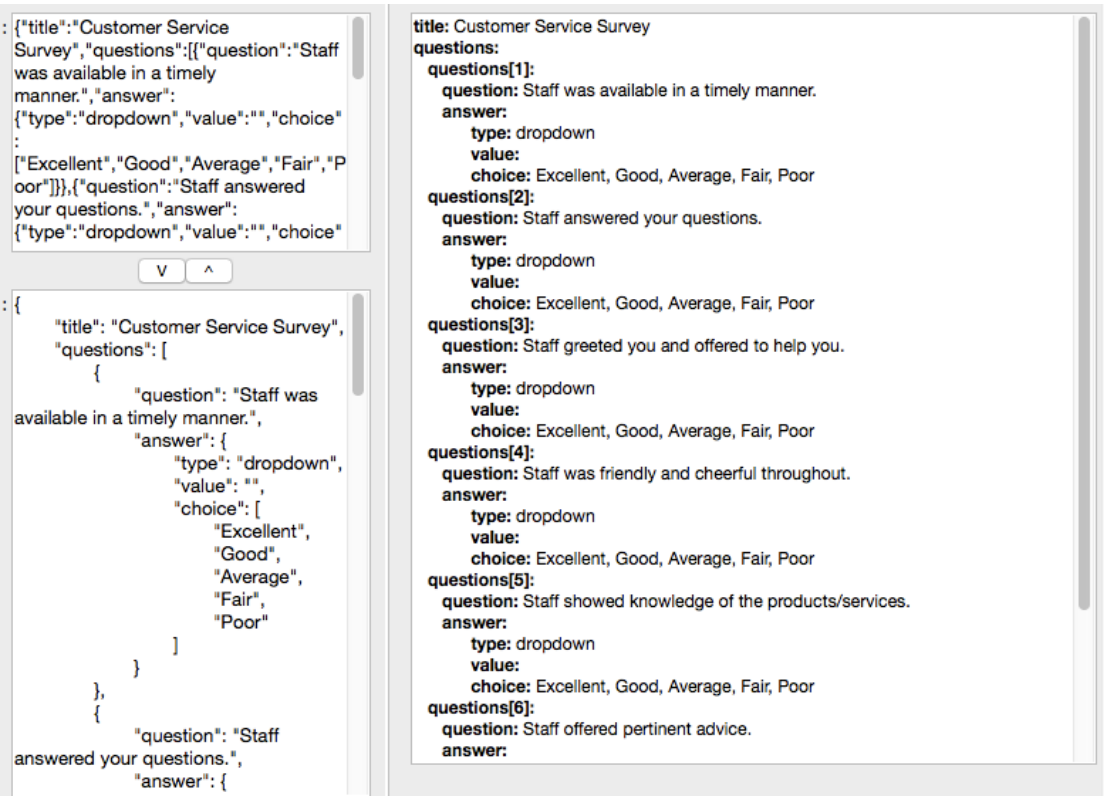
```
C_POINTER($txtPtr)
$evt:=Form event
$txtPtr:=OBJECT Get pointer(Object named;"_OBJ_TEXT_")

Case of
  : ($evt=On Load)
    C_OBJECT(myObject)
    $txtPtr->:=""

  : ($evt=On Unload)

  : ($evt=On Clicked)
    C_POINTER($nilPtr;$ptr)
    $objectPtr:=->myObject
    Util_BuildTestArrays ($objectPtr;1)
    $maxDepth:=OBJ_GetMaxDepth ($objectPtr;1;1)
    $curRef:=OBJ_SetObject2Text ($objectPtr;$txtPtr;0;0;0)
End case
```

myObject を別のオブジェクト型フィールド，あるいはオブジェクト変数に置き換えれば，その内容がマルチスタイルテキストとして表示されます（画面右側）。



表示されるテキストのアピアランスは，***OBJ_SetObject2Text*** メソッドに手を加えることにより，カスタマイズすることができます。

```
$tagBold:="<span style=\"font-weight:bold\">"
$tagItalic:="<span style=\"font-style:italic\">"
$tagUnderline:="<span style=\"text-decoration:underline\">"
$tag16Pts:="<span style=\"font-size:16pt\">"
$tagBlue:="<span style=\"color:#0000FF\">"
$tagRed:="<span style=\"color:#FF0000\">"
$tagBckgBlue:="<span style=\"background-color:#9AFFFF\">"
$tagBckgBlueBold:="<span style=\"font-weight:bold;
      background-color:#9AFFFF\">"

$tag2use:=$tagBold
```

$tag2use を変更することにより，スタイルカスタマイズすることができます。

このフォームオブジェクトは，オブジェクト型フィールド，あるいはオブジェクト変数の内容を整形して表示することができます。しかし，実際に編集することはできません。

2. DisplayObject_HL:

このフォームには，"_OBJ_HL_" という名称の階層リストとボタンが配置されており，ボタンには，下記のようなスクリプトが記述されています。

```
$evt:=Form event
Case of
    C_OBJECT(myObject)
    HL_Obj:=New list
    ck_WithPictures:=1
    ARRAY PICTURE(arr_OBJ_Icons;0)
    $path:=Get 4D folder(Current resources folder)+"images"
        +Folder separator+"4DIcons"+Folder separator
    Util_LoadIcons4Types (->arr_OBJ_Icons;$path)

  : ($evt=On Unload)
    If (Is a list(HL_Obj))
      CLEAR LIST(HL_Obj;*)
    End if

  : ($evt=On Clicked)
    C_POINTER($nilPtr;$ptr)
    $objectPtr:=->myObject
    Util_BuildTestArrays ($objectPtr;1)
    $maxDepth:=OBJ_GetMaxDepth ($objectPtr;1;1)
    $curRef:=OBJ_SetObject2List ($objectPtr;
        ->HL_Obj;0;0;->arr_OBJ_Icons;ck_WithPictures)
End case
```

お気づきのように，メソッドはスタイル付きテキストのものと非常によく似ています。違っているのは，テキストではなく階層リストが使用されている点，そしてアイコン画像が追加されている点です。

このフォームオブジェクトも，オブジェクト型フィールド，あるいはオブジェクト変数の内容を整形して表示することができますが，実際に編集することはできません。

3. DisplayObject_LB:

このフォームには， "_OBJ_LB_1" という名称のリストボックス，および "_BUTTONBAR_1" という名称のサブフォームが配置されており，そのサブフォームは "OBJ_ButtonBar" フォームを参照しています。

リストボックスなので，値を入力することができ，任意のプロパティを追加・削除・更新することができます。オブジェクトの配列など，込み入ったタイプのプロパティでも大丈夫です。ユーザーは，ドラッグ/ドロップ操作でプロパティの位置を変えることができ，オブジェクトの構造を柔軟に組み替えることができます。プロパティの編集には， "OBJ_Edit_Element" フォームが使用されています。

4D+OOPで作られたフォームオブジェクトは，まるで魔法のように移植が簡単です。

たとえば，顧客テーブルがあり，ユーザーは，独自のフィールドを自由に追加できるようになることを求めています。あなたはテーブルにオブジェクト型フィールドを追加し，"_OBJ_LB_1" リストボックスと "_BUTTONBAR_1" を入力フォームに追加します。追加したフォームオブジェクトとフィールドの関連性を定義するだけで，作業完了です。

テーブルにオブジェクト型フィールドをもうひとつ追加した場合も，"_OBJ_LB_1" および "_BUTTONBAR_1" を複製すれば，4Dが自動的に発行する新しいオブジェクト名により，最初のリストボックスと同じように使用することができます。

オブジェクトとメモリの関係を正しく理解する

ダイナミック変数

4Dで使用できる変数の級（クラス）には，ローカル変数，プロセス変数，インタープロセス変数，そして**ダイナミック変数**があります。

ローカル変数: 変数名は "$" から始まります。

定義されたメソッドの中だけで使用することができる変数です。コンパイラーは，ローカル変数のテーブルを作成し，メソッドの実行時にこの変数テーブルがスタックにロードされます。アクセスは高速で，とても効率的です。

パラメーター（$0，$1，$2...）も名前が "$" で始まりますが，ローカル変数とは違います。

ローカル変数はデータベース全体にわたって定義されているわけではありませんが，それでもポインター経由であれば参照することができます。ただし，フォームに表示させることはできません。ローカル変数が存在するのは，それを定義したメソッドが実行されている間だけなので，ローカル変数に対するポインターを使用する場合，そのメソッドが終了した後にポインターを参照することがないよう，注意する必要があります。

プロセス変数（グローバル変数とも）：変数名は 普通の文字で始まります。

データベース全体にわたって定義されますが，プロセス毎に独立した変数が存在します。コンパイラーは，グローバル変数のテーブルを作成し，アプリケーションを起動するとその変数テーブルがヒープと呼ばれるメモリ領域にロードされます。このヒープは，メインプロセスの一部です。新規プロセスが作成されると，その新規プロセスのためにテーブルが再作成されます。それで，プロセス1の変数v1は，プロセス2の変数v1とは別の値を持つことができます。

インタープロセス変数: 変数名は "<>" から始まります。

データベース全体にわたって定義され，どのプロセスからでもアクセスすることができます。コンパイラーは，インタープロセス変数のテーブルを作成し，アプリケーションを起動するとその変数テーブルが特定のプロセスに属さないヒープ領域にロードされます。プロセス1の変数<>v1とプロセス2の変数<>v1は同じものです。インタープロセス変数はフォームに表示することができ，プロセス変数を使用するようなコマンドはすべてインタープロセス変数にも対応しています。

ダイナミック変数: 変数に名前はありません。変数には，ポインター経由でアクセスします。

```
$p := OBJECT Get pointer (Object named; "_VarName_")
```

内部的には， $form.2A.4 のような変数名が付けられているかもしれませんが，この名前をプログラムの中で参照するべきではありません。

ダイナミック変数は，アクティブフォームオブジェクト（ボタン，テキスト入力など）の変数名プロパティを省略すると自動的に作成されます。そのようなオブジェクトが含まれるフォームを表示すると，インタープリターが管理しているプロセス変数の名前空間内で一意の動的な変数名が発行され，フォームオブジェクトの変数として使用されます。このメカニズムは，コンパイルモードであっても有効です。作成された変数は，フォームが閉じられると同時に破棄されます。もう一度，同じフォームを表示すれば，おそらく別の名前が発行されるはずです。コンパイルモードでダイナミック変数を使用するためには，変数の型を明確に定義する必要があります。変数の型は，下記いずれかの方法で定義します。

プロパティリストの「変数タイプ」メニューで型を設定する方法。これは単なる「希望」の表明に過ぎないことに留意してください。この操作がコンパイラーに変数の型を指示するわけではありません。

フォームが表示されるときに明示的な初期化コードを実行する方法。たとえば，VARIABLE TO VARIABLE コマンドを使用することができます。

```
C_TEXT($init)
$Ptr_object:=OBJECT Get pointer(Object named;"myDynVar")
$init:=""
VARIABLE TO VARIABLE(Current process;$Ptr_object->;$init)
```

ダイナミック変数のメカニズムを活用することには，さまざま利点があります。

まず，同じホストフォームの中で同一の「サブフォーム」コンポーネントをいくつも配置することができます。それぞれのサブフォーム内には同じ名前のオブジェクトが存在しますが，ダイナミック変数を使用していれば，変数同士が干渉することはありません。

さらに，メモリの使用量を全体的に抑えることができます。フォームオブジェクトは，普通，プロセス変数またはインタープロセス変数に関連づけられていますが，コンパイルモードの場合，すべてのプロセス変数のインスタンスがすべてのプロセスのために作成され，これにはサーバー側のプロセスも含まれます。たとえセッション中にそのフォームが一度も表示されなか

ったとしても，プロセス変数のインスタンスは一定のメモリを占有します。フォームが表示されるときにだけダイナミック変数が作成されるようにすれば，メモリの節約になります。

ダイナミック変数は，新しい CALL FORM コマンドとの相性も優れています。フォームがウインドウに表示されている限り，どんなメソッドもポインター経由でそのフォームのコンテキストで使用されている変数にアクセスすることができます。

システム変数: 4Dが管理している特別なシステム変数は，さまざまなオペレーションの成り行きをプログラムの中で見届けられるようにするために用意されています。システム変数はどれもプロセス変数であり，プロセスは自身のシステム変数にだけアクセスすることができます。

OK は，最重要のプロセス変数です。その名前から想像できるように，対象プロセスの中で物事がうまく行っていることを知らせる役目があります。タスクが成功すれば1，失敗すれば0がシステム変数 OK に代入されます。システム変数の一覧はドキュメントに掲載されています。

インタープリター専用のプロセス変数: 非常に特殊な用途ですが，インタープリターのプロセス変数だけに「生息」する定義が未定のプロセス変数を作成することもできます。型を決定する必要がないのは，変数がコンパイラー「目」に触れないためです。

正しい参照型の扱い方

4Dには，**参照**でアクセスするタイプのオブジェクトがいくつか存在します。参照は，一種のポインターであるともいえますが，4Dのポインター型とは違います。参照は，しばしば倍長整数の形を取りますが，その実体は，メモリ参照，リストのインデックスなど，さまざまです。参照の内容を直接書き換えるようなことをしてはなりません。プログラムの中で参照を失ったり，消去したりすることがないようにも注意するべきです。

ビッグオブジェクト（ピクチャ・テキスト・オブジェクト）は，ほんとうに複製する必要に迫られるまで，コピーが遅延されるようになっています。たとえば， vPict に画像が保存されている場合，vPict2:=vPict というコードを実行しただけでは，画像はコピーされません。vPict2の内容を変更したときにはじめて画像が複製され，変数に代入されて処理されます。

昔の4Dでは，おおきなオブジェクトを直接パラメーターとしてメソッドに渡すことは避けたほうが良い，とされていました。たとえば，

```
$size:=MyMethod (vPict)
```

というコードがあり，メソッドは下記のような処理をしているようなケースです。

```
$pict:=$1
$0:=Picture size ($pict)
```

ピクチャは $1 にコピーされ，次いで $pict に再度コピーされるというのが理由でした。それで，当時は代わりにポインターを使用することが推奨されていました。

```
$size := MyMethod (->vPict)
```

メソッドの内容は下記のようになります。

```
$0:=Picture size ($1->)
```

こうすれば，画像のコピーが発生しない，というわけです。

現バージョンでは，最初に挙げた方法でも画像のコピーは発生しません。メソッドは画像にまったく手を加えていないからです。

ビッグオブジェクト（BLOB）は，従来どおり，単純にコピーされています。

BLOBをメソッドに渡すのであれば，サイズに注意を払う必要があります。

```
$size := MyMethod (vMyBlob)
```

というコードがあり，メソッドは下記のような処理をしているとしましょう。

```
$blob:=$1
$0:=Blob size ($blob)
```

ピクチャは $1 にコピーされ，次いで $blob に再度コピーされます。ですから，不必要な複製を避けるため，ポインターを従来どおり使用することが勧められています。

```
$size := MyMethod (->vBlob)
```

メソッドの内容は下記のようになります。

```
$0:=Blob size ($1->)
```

こうすれば，BLOBのコピーが発生しないので効率的です。

New list，BLOB to list，Copy list，Load list といったコマンドで**階層リスト**を作成すると，リスト参照（ListRef）型とも呼ばれる倍長整数が返されます。ListRef は，メモリ内で構築さ

れたリストオブジェクトに対するポインターのようなものです。CLEAR LIST をコールするまで，作成したリストオブジェクトはメモリ内に残されています。リスト参照は，このオブジェクトにアクセスする唯一の手段なので，失くさないように注意しなければなりません。

リスト参照は，すでにポインターのようなものなので，これにポインター経由でアクセスすることには意味がありません。

```
vMyListRef:=New list
(1)MyMethod(->vMyListRef)  //Useless but harmless
(2)MyMethod(vMyListRef)  // Will work
```

この場合， MyMethod の内容は下記いずれかとなります。

Case (1) `CLEAR LIST ($1->)`
Case (2) `CLEAR LIST ($1)`

どちらも同じ意味です。ちなみに，正しくは次のようにするべきでした。

```
$listRef:=$1
If (Is a list($listRef))
  CLEAR LIST($listRef;*)
End if
```

よくある間違いは，リストの複製と関連があります。

```
vMyListRef2:=vMyListRef
```

上記のコードは，リスト参照のコピーを作成していますが，リストそのものを複製しているわけではありません。後で CLEAR LIST (vMyListRef2) を実行すると， vMyListRef が参照するリストも消滅するになります。

リストを複製するためには，コマンドを使用します。

```
vMyListRef2:=Copy list(vMyListRef)
```

もうひとつ，よくある間違いは，サブリストが接続されているリストの片付けと関連があります。第1レベルのリストをクリアしても，サブリストはそのままメモリに残されているので，サブリストの参照を管理していなかった場合，どんどんメモリに遺失物が溜まってゆきます。そのようなわけでリストをクリアするときにはサブリストも一緒にクリアするのが無難です。

```
If (Is a list($listRef))
  CLEAR LIST($listRef;*)
End if
```

メモリの取り残されたリストは，アプリケーションを終了するまでずっと残っており，利用できるメモリが徐々に減少する問題（メモリーリーク）の原因となります。

メニュー参照:

コマンドで作成されたメニューには，階層リストと同じように固有の参照があり，その参照はアプリケーション全体において有効です。MenuRef とも呼ばれるこの参照は，16桁の文字列であり，メニュー関連コマンドには，メニューまたはメニューバーを特定するため，この参照文字列を渡すことができます。

昔のバージョンでは，メニューを番号で参照していました。MenuRef は，v11以降の新しい方式です。参照を使用すれば，デザインモードのメニューエディターを使用することなく，完全にコーディングだけでメニューを作成することができ，階層メニューも扱うことができます。メニュー番号は，メニューエディター上での位置と対応しています。メニュー番号は，メニューまたはメニュー項目を特定するために使用され，メニュー管理コマンドにメニュー番号を渡した場合，カレントメニューバーが操作の対象になります。

古い方式は，現バージョンでも引き続き使用することができますが，v11以降の新しいコマンドをフル活用することができません。たとえば，階層サブメニューにはメニュー番号でアクセスすることができない，といった制約があります。

新旧の方式は，併用することができます。ほとんどメニュー関連コマンドは，MenuRef と番号の両方に対応しているからです。しかし，新しい方法のほうが明確かつ柔軟なプログラミングを可能にするので，できれば MenuRef に切り替えることが勧められています。

メニューエディターで作成されたメニューの参照は，下記のコード取得することができます。

```
$MenuRef:=Get menu bar reference{(process)}
GET MENU ITEMS($MenuRef;menuTitlesArray;menuRefsArray)
```

CREATE MENU コマンドは，メモリ内に新しいメニューを作成します。このメニューがデザインモードのメニューエディターに追加されることはありません。コマンドは， MenuRef 型の参照を返します。作成したメニューに対して加えられた変更は，即，アプリケーション内でこのメニューを参照しているすべてのプロセスのすべてのメニューに波及します。

任意のパラメーターを渡さなかった場合，新しい空のメニューが作られます。メニューに項目を追加するには，SET MENU ITEM コマンドなどを使用します。任意のパラメーターを渡した場合，指定したメニューのコピーが作られます。元のメニューにサブメニューが存在すれば，それらもすべてコピーされます。つまり，サブメニューそれぞれのコピーが作成されます。コピー元のメニューは，メニューバー番号でも指定することができます。

メニューバーを置き換えるには， SET MENU BAR コマンドを使用します。

CREATE MENU で作成したメニューは，必ず RELEASE MENU でメモリから解放することを忘れないでください。

プラグインコマンド: クイックレポート（QR New offscreen area）や 4D Write プラグイン（WR New offscreen area）などから返される倍長整数型のオフスクリーンエリア参照は，そのエリアの使用が済んだところで解放する必要があります。

Webエリア: Webエリアは，フォームエディターの「プラグイン/サブフォーム」ツールで作成することができるフォームオブジェクトです。他のオブジェクトと同様，Webエリアにもオブジェクト名と変数名があり，いずれかの方法でプログラムから参照することができます。Webエリアの変数は，テキスト型ですが，参照となる値は代入されていません。そのため，この変数をパラメーターとしてメソッドに渡すことはできません。参照ではないからです。たとえば，MyArea という変数のWebエリアを次のような方法で制御することはできません。

```
Mymethod(MyArea)
WA REFRESH CURRENT URL($1) `Does not work
```

Webエリアの変数を参照として使用するのであれば，ポインターが必要です。

```
Mymethod(->MyArea)
WA REFRESH CURRENT URL($1->) `Works
```

ドキュメント参照: 歴史的な理由により，4Dではドキュメントの参照が倍長整数ではなく時間型で返されます。

```
$vhDocRef:=Open document("";"TEXT")
```

変数 $vhDocRef は，このドキュメントを参照する手段のひとつです。もうひとつの方法として，システム変数 DOCUMENT に代入されたパスを使用することもできます。読み書きモードで開かれたドキュメントは，必ず閉じる必要があります。閉じられていないドキュメントは，

他のプロセスに対してロックされており，更新した内容は，まだディスクに書き込まれていないかもしれません。

ウインドウ参照: WinRefは，倍長整数型であり，特定のウインドウを指定するために使用されます。開かれているウインドウの参照リストは，WINDOW LIST でいつでも取得できます。

プログラムは，クリア・リリース・破棄など，それを手放すときが来るまで，原則的に参照をずっと保持しているべきです。作成したメソッドの中で使い切ることが分かっているのでない限り，参照をローカル変数に代入することは避けたほうが良いでしょう。

BLOB

バイナリ・ラージ・オブジェクト，つまりBLOBは，4Dが内容を理解しようとしない，唯一の変数タイプです。デベロッパーは，自由にデータの構造を定義し，使用することができます。BLOBの内容は，4Dによって勝手に変えられることがないので，どんなタイプのデータでも安心して保存することができます。テキストを保存するとエンコーディングが施され，ピクチャを保存すると画像データがカプセル化されることがあります。しかし，BLOBはデータがそのまま保存され，デベロッパーは思惑どおりにこれを扱うことができます。

BLOBの比較: ティボウ・アルギエール（Thibaud Arguillere）は，ブログ記事「幼少時代からの夢（Childhood Dream）」の中で配列の比較という問題について述べています。

http://www.4d.com/blog/childhood-dream.html

普通は次のような手順が思い浮かぶでしょう。

- 配列のタイプを比較する
- 合致すれば，要素の数を比較する
- 合致すれば，要素を1個ずつ比較する

BLOBを活用し，次のように最適化することができます。

```
C_BLOB($blob1;$blob2)
VARIABLE TO BLOB(MyArray1;$blob1)
VARIABLE TO BLOB(MyArray2;$blob2)
$MD5_1:=Generate digest($blob1;MD5 digest)
$MD5_2:=Generate digest($blob2;MD5 digest)
```

```
If ($MD5_1=$MD5_2)
  -> Then the 2 arrays are strictly identical...
```

データ構造体: 検索の対象にならないデータであれば，その型に関係なく，1個のBLOBに全部まとめることができます。たとえば，下記のようなメソッドをコールし，顧客のプロフィール情報をBLOBフィールド変換して保存する，というテクニックがあります。

```
$birthDate:=$1
$dogName:=$2
$carPlate:=$3
$hairColor:=$4
$housePicture:=$5
  // ... and so on...
C_BLOB($blob)
$offset:=0
VARIABLE TO BLOB($birthDate;$blob;$offset)
VARIABLE TO BLOB($dogName;$blob;$offset)
VARIABLE TO BLOB($carPlate;$blob;$offset)
VARIABLE TO BLOB($hairColor;$blob;$offset)
VARIABLE TO BLOB($housePicture;$blob;$offset)

$0:=$blob
```

このようにするメリットのひとつに，顧客テーブルを変更することなく，擬似的なフィールドを柔軟に追加することができる，という点を挙げることができます。

しかし，オブジェクト型の登場により，こうしたテクニックは過去のものとなりました。

セキュリティ: 複雑なパスワードシステムなどにより，アプリケーションに対するアクセスが厳しく制限されていたとしても，データファイルにアクセスできる人は，誰でもバイナリエディターなどで実際のデータを覗くことができます。

そのようなわけで，機密データはBLOBフィールドに保存し，さらに内容を隠匿するための処置を施すことができます。

- COMPRESS BLOB で圧縮した後，バイトスワップをかける

- ENCRYPT BLOB で暗号化する
- BLOB をピクチャーに変換してごまかす
- その他

サイズの節約: すでに圧縮されている画像は，もうそれ以上，圧縮ができないかもしれませんが，長いテキストは，BLOBに変換して圧縮すると，かなりサイズを抑えることができます。

ポインターと参照の違い

参照は，対象オブジェクトの種類により，いろいろなタイプが返されます。

4Dのポインターは，オブジェクトの参照というよりも，オブジェクトを間接的に説明するものであり，リダイレクト情報に近いものです。たとえば，変数であれば，変数名・変数スコープ・（配列であれば）要素番号・コンポーネント情報，といったデータがポインターを構成ししています。フィールドであれば，テーブル番号およびフィールド番号がこれに含まれます。

JSON-likeなオブジェクト

オブジェクト型（JSONオブジェクト，C_OBJECT）は，v14以降で登場した，新しいタイプのデータ構造体です。JSON（**J**ava**S**cript **O**bject **N**otation）は，手軽にデータを保存したり交換したりできる，XMLよりもシンプルなシンタックスです。

3名分の従業員レコードをJSONとXML，それぞれで表現して比較してみましょう。

JSON

```
{"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]}
```

XML

```
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
```

```
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
</employees>
```

JSONは，プログラミング言語に依存せず，自明的で容易に理解することができます。シンタックスは JavaScript のものですが，XMLと同様，単なるテキストであり，どんなプログラミング言語でも使用できる汎用的なデータ形式になっています。

簡単なデータ構造から連想配列まで，さまざまなオブジェクトを表現することができ，PHPでも使用できますし，当然，4Dでも使用することができます。

これはJSON構造の例です。

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    },
    "phoneNumbers": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "fax",
            "number": "646 555-4567"
```

```
        }
    ]
}
```

全体として連想配列のような構造であり， phoneNumbers のように，ある要素はそれ自体が連想配列であることに気づくと思います。{ と } に挟まれている部分は構造的な要素であり，[と] に挟まれているのは配列です。

4Dの配列は JavaScript または PHP における添字配列に相当するのに対し，JSONオブジェクトは JavaScript または PHP における連想配列になぞらえることができます。

```
$age=array("Peter"=>"35","Ben"=>"37","Joe"=>"43");
```

または

```
$age['Peter']="35";
$age['Ben']="37";
$age['Joe']="43";
```

4DのPHPにもJSONライブラリが含まれていますが，v14以降であれば，4DでJSONを扱ったほうが簡単です。

4D，PHP，JavaScript，RESTサーバーなど，さまざまなアプリケーション間でデータを交換する最良の方法は，JSONオブジェクトを使用することである，と言って良いでしょう。

JSONオブジェクトは，どのタイプの変数（ローカル，プロセス，インタープロセス）でも使用することができ，v15以降はフィールド型としても選択できるようになりました。JSONオブジェクトは，テキストに変換（Stringify）したり，テキストから復元（Parse）したりすることができます。

他にも，JSONはさまざま場面で威力を発揮します。

- メソッドに渡されるパラメーターとして
- メソッドから返される複数の戻り値として
- 構造が一定ではないレコードを保存するためのフィールドとして
- アジャイル開発において

オブジェクトに関して憶えておくべきこと

- オブジェクトは内部的に参照カウントが働いています。つまり，必要に迫られるまで，コピーされません。
- オブジェクトは参照型（一種のポインター）であり，そのような意味では階層リストに似ています。もっとも，内部的な仕組みは異なっています。
- オブジェクトの複製（OB Copy）は再帰的なコピーであり，サブオブジェクトもすべて複製されます。
- 同じオブジェクトに複数のプロセスから同時にアクセスするのは安全ではありません。
- VARIABLE TO VARIABLE，SETおよびGET PROCESS VARIABLE，New processにパラメーターとして渡されたオブジェクトは，常に複製されます。
- ポインターが含まれているオブジェクトを複製すると，ポインターもそのまま複製されますが，OB Copyに * オブションを渡した場合には，ポインターが参照しているオブジェクトの値が複製されます。ポインターの扱いには注意してください。

オブジェクトの演算

4DObjectsJPR サンプルには，オブジェクトの演算例がいくつか含まれています。

普通の変数であれば，何も難しいことはありません。数値，日付，ブール値は，いずれも簡単に比較することができます。文字列の場合，文字コードに基づいて比較するのか，それともUnicode の等価性アルゴリズムに基づいて判断するのか，あるいは単純に大文字と小文字を区別しない比較をするのか，といった選択肢があるので少しだけ話が難しくなります。

オブジェクトの比較は，それよりもずっと複雑です。たとえば，オブジェクトのプロパティ名は大文字と小文字を区別します。つまり， MyProp と Myprop は別のプロパティです。さらに，オブジェクトの構造（プロパティのリスト）だけを比較するのか，それとも内容（プロパティの値）だけ比較するのか，両方とも比較するのか，という問題もあります。

サンプルで扱っているのは，下記の演算です。

- オブジェクト内を検索
- オブジェクトの比較（構造および/または内容）

- オブジェクトにオブジェクトを加算
- オブジェクトからオブジェクトを減算
- オブジェクトの並び替え（構造および/または内容）

オブジェクトの統合（マージ）についていえば，どのようなルールで処理を進めるのか，決めなければなりません。

たとえば,

Ob_1.Name = "JPR"　&　Ob_2.Name = "Keisuke"

という演算の場合，下記どちらの結果を期待するでしょうか。

Ob_3.Name = "JPR, Keisuke"　(1 string)
Ob_3.Name = ["JPR", "Keisuke"] (String Array)

他にも，合理的なルールが考えられるかもしれません。

どれが正しいということはないので，各自の判断でコードを自由に書き換えてください。

フォームオブジェクトを効果的に管理する

オブジェクト名

フォームオブジェクトを名前で操作できるようになったことは，4Dにおける重要な転換点のひとつでした。それ以前，デベロッパーは「ckOKというチェックボックスを追加」といった表現を普通に用いていました。厳密にいえば，チェックボックスというフォームオブジェクトを作成し，そのフォームオブジェクトを ckOK という変数に結びつけていたのですが，4Dではそのあたりの概念が簡略されていました。

オブジェクトに固有の名前が設定されるようになったことにより， "_CHECK_OK_" という名前のチェックボックスを作成し，それに ckOK 変数を関連づける，という関係性がより明確になりました。もうひとつの利点として，ck_Blue，ck_Pink，ck_Red のような変数が割り当てらえているチェックボックスをまとめて操作する，といったこともできるようになりました。それぞれに "_CK_CLR_BLU_", "_CK_CLR_PINK_", "_CK_CLR_RED_" のように命名しておけば良いのです。変数で指定する場合には3行のコーディングが必要でした。

```
OBJECT SET VISIBLE(ck_Blue;False)
OBJECT SET VISIBLE(ck_Pink;False)
OBJECT SET VISIBLE(ck_Red;False)
```

オブジェクト名で指定すれば1行だけで済みます。

```
OBJECT SET VISIBLE(*;"_CK_CLR_@";False)
```

チェックボックスの値は次のような方法で取得することができます。

```
$colorName:="Red"
$flag:=OBJECT Get pointer(Object named;"_CK_CLR_"
                                        +$colorName+"_")->
```

オブジェクト名の使用を徹底する

ダイナミック変数の使用には，オブジェクト名による参照が欠かせません。これはプロセス変数テーブルのサイズを節約する（4D Server に多数のクライアントが接続し，多数のプロセスを管理しなければならないときには非常に重要）だけでなく，コンテキストに依存しないメソ

ッドや（OOP的な意味での）オブジェクトを設計し，柔軟なプログラミングを可能にする点でとても効果的です。

サンプルでは，名前ひとつを受け取るだけで状況に適応する一連のメソッドやフォームオブジェクトが使用されています。デベロッパーは，メソッドやコードブロックの内部的なメカニズムを知らなくても，自由にそのオブジェクトを使い回すことができます。

テーブル，フィールド，その他のデータベースオブジェクトについていえば，4Dデベロッパーは番号で参照することに慣れているかもしれません。しかし，これからは内部参照ではなく，名前の使用に切り替えるべきでしょう。結局のところ，内部参照は4D特有のものであり，SQL，JavaScript，PHP など，他の言語はそべてテーブル名やフィールド名を使用します。コードの可読性は高まり，内容を理解して保守することも容易になります。

```
do_Button_Action (Table(36))
```

よりも

```
do_Button_Action ("CLIENTS")
```

のほうがずっとわかりやすいのではないでしょうか。

オブジェクトとデータソース

フォームオブジェクトと，それに接続されたデータソースを混同してはいけません。オブジェクトの種類により，データソースは変数，フィールド，あるいはテーブルの場合があります。

```
OBJECT SET DATA SOURCE(*;"_CK_CLR_"+$colorName+"_";->ck_Red)
LISTBOX SET TABLE SOURCE (*;"_LB_LB1_";$tableNb)
```

リストボックスを全面的に活用する

リストボックスはまさに魔法の箱です。v11以降，もっとも重点的に強化されてきたユーザーインターフェースであり，ほんとうに多彩な面を持っています。この章では，できるだけ少ないコーディングやフォームの変更だけで画面を合理的で使いやすく，楽しいものに変える方法を紹介したいと思います。

セレクションの表示

セレクションを表示する方法には，下記の2種類が挙げられます。

古典的な方法，つまり DISPLAY SELECTION または MODIFY SELECTION を使用する方法。リストフォーム（出力）が提供するグラフィカルな機能，たとえばマーカーなどが必要な場合にはこちらを選択します。

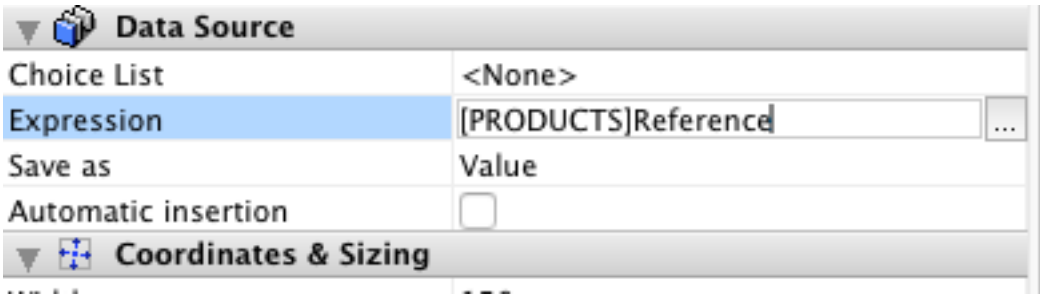
リストボックスを使用する方法。レコードの表示は4Dが自動的に管理してくれます。従来，配列を使用してセレクションを表示することにはさまざま問題が伴いましたが，リストボックスは DISPLAY SELECTION と同等の最適化がなされており，実際に表示が必要なデータだけがサーバーから転送されるので，とても 効率的です。

行・列・セルの制御

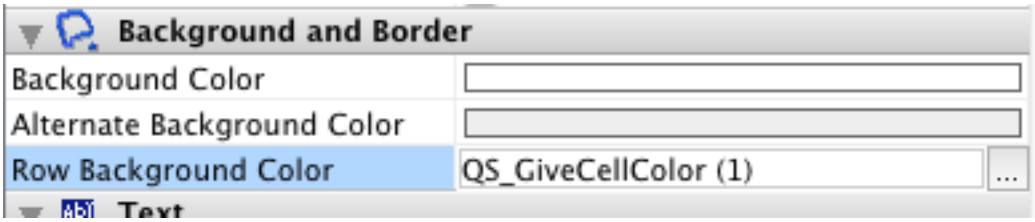
ここでもやはり，プログラミングの習慣を見直す必要があるかもしれません。これまでは，データを取り出し，加工したものを配列に代入して表示する，という考え方が普通でした。リストボックスは，もっと動的な方法で使用します。つまり，まずリストボックスにデータを表示させ，コールバックメソッドの中で必要に合わせて値を処理する，というスタイルです。

コールバックメソッドの使用

リストボックスプロパティリストには，灰色の「...」ボタンがいくつか表示されています。



もうひとつ例を挙げます。



行スタイルや行フォントカラーの場合，ボタンをクリックするとフォーミュラーエディターが表示され，任意のフォーミュラー式を入力することができます。適切な値を返すプロジェクト

メソッドの名前を入力しても構いません。v13までは，リストボックス全体のプロパティに対し，背景色や文字色を**行単位**で制御するためのコールバックメソッドを記述することしかできませんでした。v14以降，列毎にコールバックメソッド名を設定し，**セル単位**でもプロパティを制御することができます。

QS_GiveCellColor メソッドは列番号をパラメーターに受け取るメソッドです。

```
  //This Method will be called by 4D from the Row Background Color in the Background and Border area of the Property List
C_POINTER($ptr)
$columnNb:=$1  //This is the column number in the list
$ptr:=vQS_arColVars{$columnNb}
$val:=$ptr->  //This is a pointer on the field to be displayed
$color:=-255  //and the default Backround Color
If (QS_TestFit (vQS_SValue;$val))
  //Just in case you want different colors depending on the columns
  $color:=Choose($columnNb;0;0x00FFFF00;0x0066FF66;0x00FF6666)  //Yellow, green, red, etc.
End if
$0:=$color  //and we return the color in $0
```

このメソッドは，行の背景色プロパティとしてコールされることが想定されています。メソッドは，0x00RRGGBB形式の倍長整数でカラーを返し，これが対応するセルの背景色になります。リストボックスのプロパティとしてこのメソッドが使用された場合，対応する行全体の背景色が変化します。

充実したイベント管理

リストボックスには，さまざまなフォームイベントが用意されており，データ入力やクエリなどのインタフェースをカスタマイズするためにこれらを活用することができます。

100%ダイナミック

完全に動的なリストボックスを作成する方法については「"LEGO™" のようにプログラムを組み立てよう"」で取り上げます。

コンテキストについて

「"LEGO™" のようにプログラムを組み立てよう"」でも説明していることがですが，ここで鍵となるのが**コンテキスト**の概念です。フォームの実行コンテキストは，もはやプロセスと一体ではありません。1個のプロセスが同時に複数のウインドウを開くことができ，それぞれのウインドウが独自のコンテキストが持っているためです。v16では，ウインドウのコンテキストを指定して任意のメソッドを実行することができます。

```
CALL FORM($windowRef;"MyMethod";$param1;$param2)
```

これは， `$windowRef` のコンテキストで MyMethod メソッドを実行せよ，という意味です。

加えて，フォームにサブフォームが配置されているのであれば，それぞれのサブフォームにも独自のコンテキストがあります。サブフォームを正しく扱うためには，それぞれのコンテキストでどのようにイベントが発生し，どのように連鎖させられるのかを理解することが必要です。「"LEGO™" のようにプログラムを組み立てよう"」ではそのことにも触れています。

オブジェクト指向プログラミング（OOP）を4Dで実践する

Wikipediaは，OOPを次のように定義しています。

オブジェクト指向プログラミング（OOP）とは，しばしば属性とも呼ばれるフィールドにデータが収められ，しばしばメソッドとも呼ばれるプロシージャーにコードが実装されている「オブジェクト」を中心としたプログラミングの理論体系（パラダイム）である。特徴的な点として，オブジェクトのプロシージャー/メソッドは，それが関連付けられたオブジェクト自体のフィールド/属性にアクセスし，それを書き換えることができる（オブジェクトには "this" や "self" といった概念がある）。オブジェクト指向プログラミングでは，相互にメッセージを送り合うオブジェクトの集合としてプログラムを構成する。（英文記事から抄訳）

4Dのプログラミング言語は，オブジェクト指向ではありません。とはいえ，プログラミングにオブジェクト指向の方法論を取り入れることはできます。

データ，更にデータ，常にデータ…

データベースである以上，すべての物事は密接にデータと関連しています。どんなプログラミングの技法を選ぶとしても，それはデータをよく理解した上での決定であるべきです。フィールド定義・インデックス設定・メモリやディスクの使い方は，データの特性に基づいていなければなりません。いくつか例を考えてみましょう。

パラメーターの使用を徹底…

変数のスコープをしっかり管理し，オブジェクトやメソッドの独立性をできるだけ保つようにいつも心掛けることは，非常に良い方法であり，どんなプログラムでも有用です。

たとえば下記のメソッドを考慮してください。

```
vText:=""
ARRAY TEXT(myArray;50)
...
gnl_Array2Text
```

コールされているのは下記のようなメソッドです。

```
For ($i;1;Size of array(myArray))
  If ($i>1)
```

```
    vText:=vText+Char(Carriage return)+myArray$i}
  Else
    vText:=vText+myArray{$i}
  End if
End for
```

このプログラムは可視性に難点があります。メソッドを一瞥しただけでは，プロセスに一体どのような影響が発生するのか，まったく見当がつきません。メソッド名 ***gnl_Array2Text*** はある程度，内容を説明していますが，それだけでは不十分です。むしろ，次のような書き方のほうが優れています。

```
$0:=gnl_Array2Text (->$arResults)
```

コールされているのは下記のようなメソッドです。

```
C_POINTER($arPtr)
C_TEXT($result)
C_LONGINT($i)
$arPtr:=$1
$result:=""
For ($i;1;Size of array($arPtr->))
  If ($i>1)
    $result:=$result+Char(Carriage return)+$arPtr->{$i}
  Else
    $result:=$result+$arPtr->{$i}
  End if
End for
$0:=$result
```

昔の4Dでは，こうしたルールを徹底することが必ずしも容易ではありませんでした。しかし，ポインターやオブジェクト型が使用できるようになった今，最初に挙げたようなプログラミングをいつまでも続けている理由はありません。

パラメーター数が多い，パラメーター数が一定ではない，いろいろな値をメソッドから返さなければならないなど，パラメーターの扱いが複雑になりそうなときには，どうか，オブジェクト型のパラメーターを積極的に活用してください。

プログラミングの最適化

いろいろな方法でプログラミングを改善することができるかもしれません。しかし，最適化に取り掛かる前に，それが必要かどうか，自問してみるのは良いことです。最適化に投じなければならないコストとそこから得られる実質的なメリットを考量してください。数ミリ秒の高速化は，それが数百万回のループに関わるのでない限り，あまり意味がありません。ですから，プログラミングの最適化は，次の条件が満たされているときにだけ実施するべきです。

- ほんとうに必要に迫られている

- コードの可読性を低下させない

- 新しい問題を作り出さない

「ボイヤール要塞」の教訓を銘記しましょう。

必要があり，簡単で，無害で，安全な場合にだけ，最適化する，ということです。

配列処理の最適化

年々，メモリは安価になっており，普通はユーザーが必要とするよりも容量に余裕があるものなので，セレクションよりも配列でプログラミングをしたほうが便利なように思えるかもしれません。しかし，注意していないと，数百万件の巨大な配列ができることもあります。Find in array は1万要素の配列では高速ですが，500万件ではずっと遅くなります。特に要素が配列の最後のほうに位置していたり，配列内に存在しない場合にはそれが顕著です。

- 並び替え済み配列

可能な場合には，事前に配列を並び替えておくことができます。Find in sorted array は，違うアルゴリズムを採用しており，通常の配列サーチよりもずっと高速です。バイナリサーチは，二分探索また分割統治法に分類され，対数時間で完了することができます。もちろん，配列の並び替えには相応の時間が必要なので，ある程度の回数，同じ配列を検索することが前提になります。

- 並び替えてはいけない配列

配列の並び替えが実際的ではない場合もあります。あるいは，レコード番号やブロック番号など，要素番号に特別な意味があり，それを保持する必要があるかもしれません。そのような場

合，きちんとした値を返すような検索の結果を別の配列にキープしておき，一種のキャッシュメカニズムを使用することができます。あるいは，もとの要素番号が代入された第2の配列を用意し，マルチソートを実行することにより，いつでも並び替え前の位置が調べられるようにすることもできます。

- ハッシュ関数と配列

ハッシュとは，なんであれ，サイズが一定ではないデータを固定サイズのデータにマッピングすることができる関数のことです。ハッシュ関数から返される値はハッシュ値，ハッシュコード，ハッシュサム，あるいは単純にハッシュと呼ばれます。ソフトウェアにおける代表的な用途のひとつに，データの素早いルックアップに使用されるハッシュテーブルが挙げられます。ハッシュは，テーブルまたはデータベース内における重複レコードの検出を高速にすることができます。たとえば，100万件の要素を有する配列を1,000 × 1,000要素の2次元配列に置き換える例を考えてみましょう。占有するメモリはほとんど変わりません。ちょうど良いハッシュ関数をみつけるのがコツですが，ここではシンプルに Mod 関数を使用します。仮に100万個のランダムな数値があったとします。これを1,000 × 1,000要素の2次元配列に代入するとき，下記のようにハッシュ関数を使用します。

```
$colNb:=Mod(theNumber;1000)
APPEND TO ARRAY(MyArray{$colNb};$myNumber)
```

データを取り出すときにもハッシュ値を使用します。

```
$found:=Find in array(myArray{Mod($value2Find;1000)};$value2Find)
```

言うまでもなく，もっと高度なハッシュ関数を使用することもできます。冒頭で述べたように，データを理解し，データに応じた処理をすることが重要です。文字列，氏名，電話番号など，値の分布が一律か，それとも偏在しているか，といったことも考慮する必要があります。

- 連想配列

コンピューター用語として言及される場合，連想配列・マップ・シンボルテーブル・辞書はいずれも「鍵・値」ペアの集合からなる抽象的なデータ型を指します。鍵となるものは一意であり，特定の鍵は集合の中で1回だけ出現します。

マルチマップは，連想配列をより汎用的にしたもので，1個に鍵に対し，複数の値を関連づけることができます。双方向マップは，連想配列に似ている抽象的なデータ型ですが，一意の鍵が値とバインドされているだけでなく，値から一意の鍵をルックアップすることもできます。

4Dでは，JSON型のオブジェクトを連想配列のように使用することができます。従来であれば，対称的な配列を用意し，一方には商品コード，他方には商品名を代入する，といったことが普通でした。その場合，商品コードから商品名を特定する場合，まず一方の配列をサーチします。

```
$k:=Find in array($arCodes;$myCode)
If ($k>0)
  $arNames{$k}:=$myName
Else
  APPEND TO ARRAY($arCodes;$myCode)
  APPEND TO ARRAY($arNames;$myName)
End if
```

続いて他方の配列から商品名を取り出します。

```
$k:=Find in array($arCodes;$myCode)
If ($k>0)
  $myName:=$arNames{$k}
Else
  $myName:=""
End if
```

オブジェクトを使用すれば，ずっと簡単です。

```
C_OBJECT($myArray)
```

商品コードと商品名は一緒に管理することができます。

```
OB SET($myArray;myCode;$myName)
```

検索は下記の行だけで済みます。

```
$myName:=OB Get($myArray;$myCode}
```

とてもシンプルです。

ストラクチャに依存しないプログラミング

大規模なアプリケーションのメンテナンス進行（あるいは継続的メンテナンス）は，プログラミングをした人が無計画で，あまり考えずにコードを書いている場合，とても大変です。フィールドやテーブルを追加しただけで，無数のメソッドやデータ構造が変わってしまうので，検証とデバッグを完全にやり直す必要があります。

ストラクチャ定義は，常に4Dのメモリに置かれており，そのため4Dのストラクチャアクセスコマンドはどれも非常に高速です。デベロッパーは，特定のフィールドが存在すること，あるいはフィールドが特定のデータ型であることを前提にするのではなく，ストラクチャの現状に合わせて柔軟に対応するようなコードを記述することができます。

私が強い勧めたいのは，名前指向のプログラミングです。[Clients]Name のようなコードをメソッドに記述するのではなく，"Clients" というテーブル名と "Name" というフィールド名に基づき，このフィールドに対するポインターを取得します。値は，フィールドのタイプに合わせて代入します。例題 "SubForms" を参照してください。

このテクニックは，コードを再利用することができ，ストラクチャ間で容易に転用できる，という点が優れています。

ストラクチャに依存しないプログラミングのサンプルが "4DOOP" に含まれています。メソッド "EXP_DB_PropagateSelection" は，どんなストラクチャでも使用することができ，テーブル名を渡せば，波及的にリレートテーブルのセレクションを作成するようになっています。

再利用できるオブジェクト

アプリケーションが複雑になれば，開発とメンテナンスのコストは，上昇してゆきます。これを回避する，あるいは少なくとも抑えるためには，コードやオブジェクトの再利用を促進し，効率を高めるしかありません。デベロッパーは，たとえ単一のアプリケーション内であっても，"ツール" や "サービス" を考案することに力点を置き，単発の "メソッド" を作成する習性から脱却する必要があります。たとえば，2件のレコードを統合するメソッドを書く代わりに，Merging Service を設計したり，汎用的な Merging Tool を作成します。直接，クエリを記述するのではなく，Query Tool を作成するか，Query Service を用意します。こうすることには，いろいろなアドバンテージがあります。

- ツールは，いろんな場面で使用することができ，それぞれの用途に向けて投じた開発努力からすべての使用箇所が益を受けられます。

- デバッグは1回だけで済みます。
- 4Dの新しい機能や変更点に対応するのは1箇所だけで済みます。
- データにアクセスする他の方法（SQL, REST, etc.）に対応するのも1箇所だけです。

クエリ画面，プリントツールなど，フォームオブジェクトについても同じことがいえます。

"LEGO™" のようにプログラムを組み立てよう

サンプル "4DOOP" が例題です。

4Dの歴史上，もっとも重要な進歩のひとつに，フォームオブジェクトを名前で参照できるようになったことが挙げられます。それまでデベロッパーは普通に「ckOKというチェックボックスを追加した」といったことを口にしていました。しかし，実際には，チェックボックスというフォームオブジェクトを作成し，そのフォームオブジェクトを ckOK という変数に結びつけていたのです。結果的に，ユーザーがチェックボックスをクリックすると，4Dが変数 ckOK を更新するようになりました。

現バージョンでは，チェックボックスを作成し， "_CHECK_OK_" というオブジェクト名を与えておき，後で ckOK 変数と関連づける，といったこともできます。

サンプルは，簡単な請求書データベースに基づいており，Explorer というプロジェクトフォームが追加されています。ほかにも，SUB_FieldWithText, List_Tools, および Page_Tools というフォームが使用されています。Explorer が使用するメソッドには，すべて "EXP_" という接頭辞が付されています。

この Explorer モジュールは，インタープロセス変数をまったく使用しておらず，プロセス変数もオブジェクト型が1個，ユーザーの設定を保存するために使用されているだけです。このエクスプローラー画面には，複数のテーブル（Products・ Clients・Invoices）がまとめて表示されており，ユーザーはリスト画面と詳細画面を同時に閲覧することができます。表示されるフィールドはカスタマイズすることができ，リストの1行をクリックすると，他のリストに表示されるセレクションの表示も連動して更新されるようになっています。

このフォームは，すべてダイナミック変数・サブフォーム（ウィジェット）・リストボックスに組み合わせて構成されており，すべてが動的にセットアップされています。フォームには，必要なオブジェクトがパーツとして配置されており，それらが必要に応じて複製され，使用さ

れるようになっています。テーブルはデータベース定義を解析（非表示テーブルは除外）して特定され，リレーション定義に基づいて各リストボックスがセットアップされています。

Explorer のフォームメソッドは，はじめにさまざまな設定や変数を保持するためのオブジェクト変数 explorer_CurrentView を定義しています。この変数は，フォームの「記憶装置」に相当するといっても良いでしょう。加えて[Parameters]Object_Content フィールドもオブジェクトとして配置されています。目的は，各ユーザー毎の「ビュー」を管理するためですが，サンプルでは "USER_VIEW" というレコードが1件，登録されているだけなので，特にビューの変化はありません。必要に応じ，レコードを増やすことができます。

フォームには，必要なオブジェクトが配置されており，どれも名前に "_SRC_" が含まれています。フォームの On Load イベントでは，最後の行により，すべてのオブジェクトを非表示にされています。

```
C_LONGINT($tableNum;$row;$where)
$evt:=Form event
Case of
  :($evt=On Load)
```

プロセス変数を使用しても構いませんが，カスタム定数を定義しても良いでしょう。

```
k_MaxFieldsInPage:=25
```

続く部分では，ストラクチャ定義を解析し，非表示に設定されていないテーブルの名前を取得しています。別の条件，たとえばテーブル名がXXで始まる，EXPを含む，ストラクチャエディター上のカラーが青である，といった独自の条件でテーブルを除外することもできます。フォームメソッドには，ボタンの代わりにポップアップメニューを使用する場合のコード例も含まれています。

```
ARRAY TEXT($ar_TableList;0)
For ($iTable;1;Get last table number)
  If (Is table number valid($iTable))
    GET TABLE PROPERTIES($iTable;$flInvisible)
    If (Not($flInvisible))
      APPEND TO ARRAY($ar_TableList;Table name($iTable))
    End if
  End if
```

```
End for
```

それぞれのテーブルにつき，冒頭の2フィールドを用いてデフォルト設定を定義します。ここも，自由にカスタマイズできる部分です。

```
For ($i;1;Size of array($ar_TableList))
  $tableNum:=Table(EXP_Util_GetTablePtrFmName ($ar_TableList{$i}))
  ARRAY TEXT($ar_FieldsInList;0)
  APPEND TO ARRAY($ar_FieldsInList;Field name($tableNum;1))
  APPEND TO ARRAY($ar_FieldsInList;Field name($tableNum;2))
  OB SET ARRAY ($parm_object;
      "FIELDS_LIST_"+String($i);$ar_FieldsInList)
  ARRAY TEXT($ar_FieldsInPage;0)
  APPEND TO ARRAY($ar_FieldsInPage;Field name($tableNum;1))
  APPEND TO ARRAY($ar_FieldsInPage;Field name($tableNum;2))
  OB SET ARRAY($parm_object;
      "FIELDS_PAGE_"+String($i);$ar_FieldsInPage)
End for
```

メソッド ***EXP_Util_GetTablePtrFmName*** は，指定された名前を持つテーブルに対するポインターを返します。

カレントの表示設定は，変数 explorer_CurrentView に保存されています。

```
OB SET(explorer_CurrentView;"TABLEVIEW_1";"")
```

これは「テーブル1はリスト表示」という意味です。ページ表示であれば，

```
OB SET(explorer_CurrentView;"TABLEVIEW_1";"P")
```

という記述になります。

ユーザー設定は，レコードに保存されています。

```
[Parameters]Object_Content:=$parm_object
SAVE RECORD([Parameters])
```

_SRC_LIST_BUTTONS_ , _SRC_PAGE_BUTTONS_ などの名前が付けられているオブジェクトは，ひな型に過ぎないので，すべて非表示にされています。

```
OBJECT SET VISIBLE(*;"@_SRC_@";False)
```

Explorer フォームは，エクスプローラーと設定画面の2ページで構成されています。別のデータベースに移植した場合，設定する必要があるのは，画面上部のボタンだけです。どのボタンも変数名は空欄であり，したがってフォーム表示と同時に作成されるダイナミック変数です。オブジェクト名は，次のようなルールに従って付けられています。

"_SELECTOR_テーブル名_" （例: "_SELECTOR_PRODUCTS_"）

オブジェクト名に仕掛けがあることにお気づきでしょうか。どのボタンにも同じようなスクリプトが記述されています。

```
$evt:=Form event

C_POINTER($ptr)

Case of
  : ($evt=On Load)

  : ($evt=On Clicked)
    $ptr:=OBJECT Get pointer(Object current)

    $name:=OBJECT Get name(Object current)
    $name:=Replace string($name;"_SELECTOR_";"")
    $name:=Substring($name;1;Length($name)-1)

    EXP_Explorer_ShowTable ($ptr;$name;$evt)

End case
```

ハイライトされているのは，オブジェクトが自分を参照し，自分の名前からテーブル名を取り出している部分です。その後，汎用的なメソッドである ***EXP_Explorer_ShowTable*** をコールし，データを処理しています。

EXP_Util_GetTablePtrFmName あるいは ***EXP_Util_GetFieldPtrFmName*** といったユーティリティは，名前どおりのことをするメソッドです。説明は不要でしょう。

EXP_Explorer_ShowTable メソッドは，テーブルの内容をリストまたはページ形式で表示します。画面の幅が許す限り，何個でもテーブルを表示することができます。

メソッドは，前述したボタンのメソッドより，下記のパラメーターを受け取ります。

```
$ptr:=$1
$tableName:=$2
$evt:=$3
```

ここに挙げる値は，デザインに合わせて調整できるかもしれません。オブジェクト外側の余白，スプリッターの幅，スプリッターのデフォルト名などが定義されています。

```
$k_borderOffset:=15  //Here we define some 'constants'
$k_splitterWidth:=8
$defaultSplitterName:="_SRC_SPLIT_"
```

第1パラメーターはオブジェクトにバインドされている変数の値を示しています。チェックボックスであれば，0（テーブルは非表示）または1 （テーブルを表示）が返されます。

```
$currentState:=$ptr->  //0 = not visible, 1 = visible

$buttonName:=OBJECT Get name(Object current)

$tablePtr:=EXP_Util_GetTablePtrFmName ($tableName)
Case of
  : ($evt=On Clicked)  //Click on a topleft button
    ALL RECORDS($tablePtr->)

  : ($evt<0)  //Just for redraw, do not change the current selection

Else

End case
```

区分の幅を計算するためには，表示されるテーブルの数や順番を知る必要があります。まず，表示しても良いテーブルの名前をすべて取得します。

```
ARRAY TEXT($ar_PossibleDisplayedLists;0)
```

さらに，表示することが求められているテーブルの名前を取得します。

```
ARRAY TEXT($ar_CurrentDisplayedLists;0)
$count:=EXP_Util_FindAllLists (->$ar_PossibleDisplayedLists;->
$ar_CurrentDisplayedLists)
```

すべてのスプリッターを隠した後，必要なパーツ（メッセージ関連）だけを表示します。

```
OBJECT SET VISIBLE(*;"@_SPLIT_@";False)

If ($count=0)

  OBJECT SET VISIBLE(*;"@_OBJ_@";False)
  OBJECT SET VISIBLE(*;"_MESSAGE_";True)

Else

  OBJECT SET VISIBLE(*;"_MESSAGE_";False)
```

ここまで来れば，各カラムの幅を計算することができます。

```
  GET WINDOW RECT($WL;$WT;$WR;$WB)
  $k_splitterSpace:=$k_borderOffset+$k_splitterWidth+$k_borderOffset
  $splitterSpace:=$k_splitterSpace*($count-1)*Num($count>1)
  $availWidth:=($WR-$WL)-($k_borderOffset*2)-$splitterSpace
  $tableWidth:=$availWidth\$count
  $curPosition:=$k_borderOffset
```

変数 $curPosition は次のオブジェクトを描画すべき位置を示しています。値は，フォーム左端からの距離です。

使用されていないリストがあるかもしれないため，まずは全部を非表示にします。

```
  OBJECT SET VISIBLE(*;"@_OBJ_LIST_@";False)
```

ページ表示用のオブジェクトも同じ理由で非表示にします。

```
  OBJECT SET VISIBLE(*;"@_OBJ_PAGE_@";False)
```

```
  For ($i;1;$count)
```

表示するテーブルが2個以上あれば，リストやページの間にスプリッターが必要です。

```
    If ($i>1)
```

これから説明するのは，このメソッドの中でも**非常に重要な部分**です。この処理は，オブジェクトを複製するたびに実行されます。前述したように，オブジェクトの主要な識別子はその名前です。オブジェクト名が重複することはありません。フォームオブジェクトは，既存のオブジェクトを複製することにより，作成することができますが，その後，削除することはできないので，再利用（リサイクル）する必要があります。まず，利用できるオブジェクトがすでに存在するかチェックするために，ポインターの取得を試みます。ポインターが有効であれば既存のオブジェクトを再利用し，そうでなければ新しくオブジェクトを作成します。

```
      $splitterName:=$defaultSplitterName+String($i-1)
      //the name will be _SRC_SPLIT_1, 2, etc.
      $ptr:=OBJECT Get pointer(Object named;$splitterName)
      If (Not(Nil($ptr)))
          //The object exists already
      Else
        OBJECT DUPLICATE(*;$defaultSplitterName;$splitterName)
            //Object doesn't exist, we duplicate the original
        $ptr:=OBJECT Get pointer(Object named;$splitterName)
      End if
```

以後のコードは，すでにオブジェクトが存在するという前提で処理が進められます。まずは，適切な位置にオブジェクトを移動します。

```
      OBJECT GET COORDINATES($ptr->;$L;$T;$R;$B)
      $curPosition:=$curPosition+$k_borderOffset
      OBJECT SET COORDINATES($ptr->;$curPosition;$T;$curPosition+$R-
$L;$B)
      OBJECT SET VISIBLE($ptr->;True)
      $curPosition:=$curPosition+$k_splitterWidth+$k_borderOffset
    End if
```

スプリッターが配置されたところでテーブル $tableName の内容をリストまたはページ形式で描画します。

```
    $tableName:=$ar_CurrentDisplayedLists{$i}
    $curIndex:=EXP_Util_FindTableIndex ($tableName)
    Case of
      : (OB Get(explorer_CurrentView;
          "TABLEVIEW_"+String($curIndex))="P") (Therefore we use a Page)

        $curPosition:=EXP_Util_DisplayPage ($tableName;
    "_SRC_PAGE_SUBFORM_"; (Name of the Source Subform object in the form)
    "_SRC_PAGE_BUTTONS_"; (Name of the Source Button Palette in the form)
    $curPosition;     (Where to draw it)
    $tableWidth;    (Available Width)
    "_PAGEFIELD_FIELD_";(Name of the Objects in the SubForm)
    "_PAGEFIELD_TEXT_")

      Else     (Therefore we use a List)
        $curPosition:=EXP_Util_DisplayList ($tableName;
    "_SRC_LIST_";(Name of the Source List object in the form)
    "_SRC_LIST_BUTTONS_";(Name of the Source Button Palette in the form)
    $curPosition;     (Where to draw it)
    $tableWidth)    (Available Width)

      End case
    End for
  End if
  REDRAW WINDOW
```

リストを描画しているのは ***EXP_Util_DisplayList*** メソッドです。

```
$tableName:=$1
$listSourceName:=$2
$listButtonsName:=$3
$curPosition:=$4
$tableWidth:=$5

$tablePtr:=EXP_Util_GetTablePtrFmName ($tableName)
$tableNum:=Table($tablePtr)
```

ひな型のオブジェクトは，どれも名前が "_SRC_" で始まっています。オブジェクトを複製するときには， "_SRC_" を "_OBJ_" に置換した上でテーブル名を追加しています。たとえば "_SRC_LIST_" は "_OBJ_LIST_PRODUCTS_" となります。

```
$listName:=Replace string($listSourceName;"_SRC_";
                                          "_OBJ_")+$tableName+"_"
```

作成あるいは再利用されたオブジェクトの扱いはすでに説明したとおりです。

```
$listPtr:=OBJECT Get pointer(Object named;$listName)
If (Not(Nil($listPtr)))
  OBJECT SET VISIBLE(*;$listName;True)
Else
  OBJECT DUPLICATE(*;$listSourceName;$listName)
  $listPtr:=OBJECT Get pointer(Object named;$listName)
End if

OBJECT GET COORDINATES($listPtr->;$L;$T;$R;$B)
OBJECT SET COORDINATES($listPtr->;$curPosition;$T;$curPosition+
$tableWidth;$B)
OBJECT SET VISIBLE($listPtr->;True)
$originalCurPosition:=$curPosition
$curPosition:=$curPosition+$tableWidth
```

ここでリストの準備に取り掛かることができます。

```
LISTBOX DELETE COLUMN($listPtr->;1;32000)
$setName:="UserSet_"+$tableName
```

配列型ではなく，セレクション型のリストボックスなので，ソーステーブルをセットします。

```
LISTBOX SET TABLE SOURCE($listPtr->;$tableNum;$setName)

$arrayIndex:=EXP_Util_FindTableIndex ($tableName)
If ($arrayIndex>0)
  ARRAY TEXT($ar_FieldsInList;0)
  OB GET ARRAY([Parameters]Object_Content;
```

```
      "FIELDS_LIST_"+String($arrayIndex);$ar_FieldsInList)
  For ($i;1;Size of array($ar_FieldsInList))
    $fieldName:=$ar_FieldsInList{$i}
    $fieldPtr:=EXP_Util_GetFieldPtrFmName ($fieldName;"";$tablePtr)
    $titleName:="H_"+$tableName+"_"+$fieldName
    LISTBOX INSERT COLUMN($listPtr->;$i;$fieldName;
                                $fieldPtr->;$titleName;$nilPtr)
    $ptr:=OBJECT Get pointer(Object named;$titleName)
    OBJECT SET TITLE($ptr->;$fieldName)
  End for
End if
```

リストの描画は以上ですが，まだツールバーの管理が残っています。ツールバーはリスト用とページ用の2個が存在します。

```
EXP_DisplayToolBar ($listButtonsName;$tableName;$originalCurPosition)
```

ツールバーに情報を表示するのは次のメソッドです。

```
EXP_DB_DisplayInfos ($tableName;"LIST")
```

次に表示されるテーブルのために，位置情報を返します。

```
$0:=$curPosition
```

ページ形式の表示は，リストよりも少しだけ細かいことをしています。大部分のメソッドは同じものですが，リストボックスの代わりにサブフォームを複製している点が違います。

```
ARRAY TEXT($ar_FieldsInPage;0)
OB GET ARRAY([Parameters]Object_Content;
      "FIELDS_PAGE_"+String($arrayIndex);$ar_FieldsInPage)

For ($i;1;Size of array($ar_FieldsInPage))
  $fieldName:=$ar_FieldsInPage{$i}
$fieldPtr:=EXP_Util_GetFieldPtrFmName ($fieldName;"";$tablePtr)
$newSubFormName:=$subFormDefaultName+String($i)

  $moreVOffset:=EXP_ExecSF_DisplayField ($i;$tableNum;
$pageSubformName;$newSubFormName;$fieldPtr;$moreVOffset;
```

```
$pageFieldName;$pageTextName;$curPosition;$tableWidth)

End for
```

余分に作られたサブフォームは非表示にしておきます。

```
For ($i;Size of array($ar_FieldsInPage)+1;k_MaxFieldsInPage)
  $subformName:=$subFormDefaultName+String($i)
  $ptr:=OBJECT Get pointer(Object named;$subformName)
  If (($ptr=$nilPtr) | Nil($ptr))
  Else
    OBJECT SET VISIBLE(*;$subformName;False)
  End if
End for

  //Now we have to handle the toolbar
EXP_DisplayToolBar ($pageButtonsName;$tableName;$originalCurPosition)
EXP_DB_DisplayInfos ($tableName;"PAGE")
$curPosition:=$curPosition+$tableWidth
$0:=$curPosition
```

サブフォームを複製しているのは ***EXP_ExecSF_DisplayField*** メソッドです。サブフォームは，その中にテキスト・変数・フィールドなどのオブジェクトを抱えており，それぞれが固有のオブジェクト名を持っています。"_SRC_PAGE_SUBFORM_" サブフォームの複製により，"_OBJ_PAGE_SUBFORM_PRODUCTS_n" （nはフィールド番号）が作られた場合，それぞれのサブフォームに "_PAGEFIELD_TEXT_", "_PAGEFIELD_FIELD_TXT_", "_PAGEFIELD_FIELD_PICT_" といったオブジェクトが含まれることになります。これらの内部オブジェクトには，メインのフォームから名前でアクセスすることができません。そこでサブフォームのコンテキストからアクセスすることになります。

```
$ptr:=OBJECT Get pointer(Object named;$newSubFormName)
OBJECT GET COORDINATES(*;$subFormDefaultName;$L;$T;$R;$B)

If (Not(Nil($ptr)))
  //The subform object exists already
Else
  OBJECT DUPLICATE(*;$subFormDefaultName;$newSubFormName)
```

```
  $ptr:=OBJECT Get pointer(Object named;$newSubFormName)
End if

$flOK:=False
EXECUTE METHOD IN SUBFORM($newSubFormName;"EXP_SubForm_SetField"
         ;$flOK;$tableNum;$fieldPtr;$pageFieldName;$pageTextName)
```

最後の1行に注目してください。メソッド ***EXP_SubForm_SetField*** は，サブフォームのインスタンス毎に，そのサブフォームのコンテキストで実行されます。

もうひとつのメソッド ***EXP_SubForm_SetFormat*** も同じようにサブフォームのコンテキストで実行されています。

```
$flOK:=False
EXECUTE METHOD IN SUBFORM($newSubFormName;"EXP_SubForm_SetFormat";
                          $flOK;$tableNum;$fieldPtr;$pageFieldName)
```

メソッド ***EXP_SubForm_SetField*** は，サブフォーム内のオブジェクトを更新するためのものです。同じオブジェクトにテキストとピクチャを表示することはできないため，両方をオブジェクトを用意し，フィールドのタイプに合わせて切り替えています。

```
$tableNum:=$1
$fieldPtr:=$2
$pageFieldName:=$3
$pageTextName:=$4

$ok:=False
$fieldName:=Field name($fieldPtr)
$pageFieldName_Txt:=$pageFieldName+"TXT_"   (If it's any text)
$pageFieldName_Pict:=$pageFieldName+"PICT_"   (If it's a Picture)

$type:=Type($fieldPtr->)

Case of
  : ($type=Is picture)
    OBJECT SET VISIBLE(*;$pageFieldName_Txt;False)
    OBJECT SET VISIBLE(*;$pageFieldName_Pict;True)
```

```
    $pageFieldName:=$pageFieldName_Pict
  Else
    OBJECT SET VISIBLE(*;$pageFieldName_Txt;True)
    OBJECT SET VISIBLE(*;$pageFieldName_Pict;False)
    $pageFieldName:=$pageFieldName_Txt
End case

$ptr:=OBJECT Get pointer(Object named;$pageFieldName)
If (Not(Nil($ptr)))  //If the object exists...
```

オブジェクトのデータソースとしてフィールドを指定します。

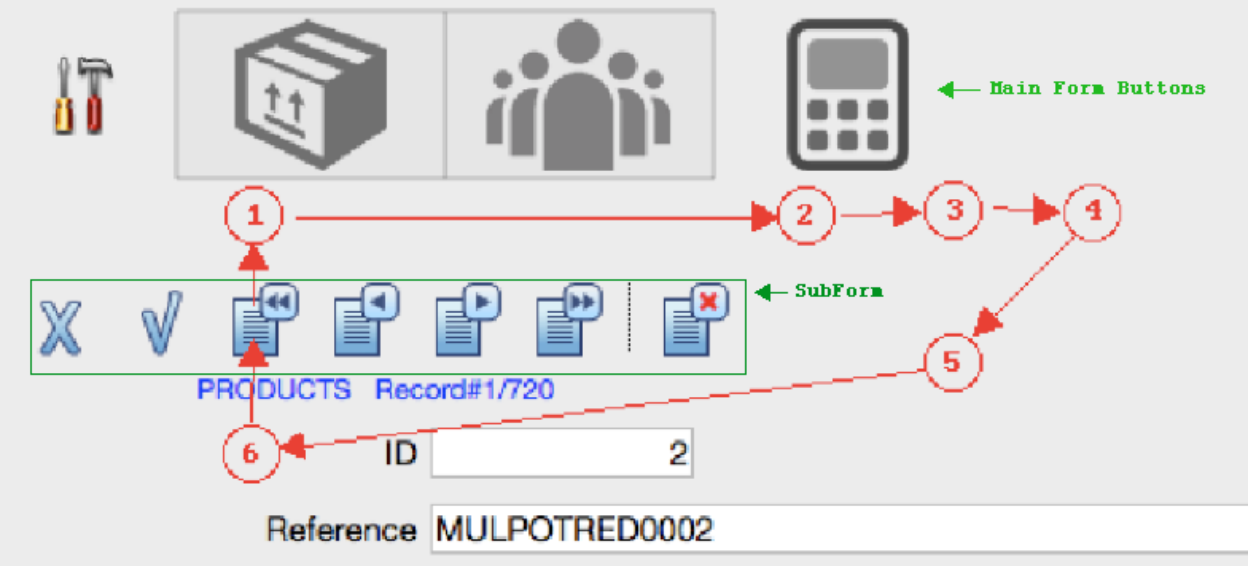
```
  OBJECT SET DATA SOURCE($ptr->;$fieldPtr)
```

最後に値を代入します。

```
  $ptr:=OBJECT Get pointer(Object named;$pageTextName)
  If (Not(Nil($ptr)))  //...and if it exists...
    $ptr->:=$fieldName  //...we set the value
    $ok:=True
  End if
End if

$0:=$ok  // We don't actually need it, but it is a training database
anyway...
```

メインのフォームに配置されたボタンのクリック処理はとても簡単ですが，サブフォームの内部に配置されたボタンの処理はちょっと複雑です。



ステップ1: ボタンがクリックされました。ボタンのスクリプトは下記のとおりです。

```
EXP_HandleEventOnToolBar (Self;Form event)
```

ステップ2: メソッド ***EXP_HandleEventOnToolBar*** がコールされます。

```
$self:=$1
$evt:=$2
$buttonName:=OBJECT Get name(Object current)
```

ボタンのオブジェクト名より，ソース（リストあるいはページ）と実行するべきアクションが判別できます。ボタンのオブジェクト名は， "_LIST_BUTTON_QUERY_", SUBSET_, ALL_, ADD_, etc. のように定義されているからです。

```
Case of
   : ($buttonName="@_LIST_BUTTON_@")
     $buttonAction:=Replace string($buttonName;"_LIST_BUTTON_";"")

   : ($buttonName="@_PAGE_BUTTON_@")
     $buttonAction:=Replace string($buttonName;"_PAGE_BUTTON_";"")

End case
```

```
$buttonAction:=Substring($buttonAction;1;Length($buttonAction)-1)
$pushedEvent:=-100
```

負のカスタムフォームイベント番号を返しているのは，既定イベントと競合しないためです。

ここでサブフォームに割り当てられた変数に対するポインターを取得します。

```
$subformContainerPtr:=OBJECT Get pointer(Object subform container)
```

その変数にアクションを説明するテキストを代入します。

```
$subformContainerPtr->:=$buttonAction
```

後はサブフォームのメソッドに処理を委ねます。サブフォームのメソッドは，メインのフォームと同じコンテキストで実行されます。

```
If ($pushedEvent<0)
  CALL SUBFORM CONTAINER($pushedEvent)
End if
```

ステップ3: ここからはサブフォームのコンテナに制御が移っています。

```
$pushedEvent:=Form event
$buttonAction:=""
$containerName:=OBJECT Get name(Object current)
$containerVarPtr:=OBJECT Get pointer(Object current)
$buttonAction:=$containerVarPtr->
```

アクションは，サブフォームの変数に代入されています。

```
Case of
  : ($pushedEvent<0)
```

フォームイベント番号が負の値だった場合にのみ，処理を続けます。通常のイベント，たとえば On Clicked などは除外する必要があります。

```
    If ($buttonAction#"")
      EXP_ExecMethodOnToolBar ($containerName;$buttonAction)
    End if
  Else
```

```
End case
```

ステップ4: メソッド ***EXP_ExecMethodOnToolBar*** が適切なアクションをテーブルに対して実行します。テーブル名は，サブフォームコンテナのオブジェクト名から特定できます。

```
$containerName:=$1
$buttonAction:=$2

If (Position("_PAGE_";$containerName)>0)
  $fl_IsPage:=True
  $tableName:=Replace string($containerName;"_OBJ_PAGE_BUTTONS_";"")
Else
  $fl_IsPage:=False
  $tableName:=Replace string($containerName;"_OBJ_LIST_BUTTONS_";"")
End if

$tableName:=Substring($tableName;1;Length($tableName)-1)
$tablePtr:=EXP_Util_GetTablePtrFmName ($tableName)
$arrayIndex:=EXP_Util_FindTableIndex ($tableName)

$fl_RefreshInfos:=False
$fl_Redraw:=False
Case of
  : ($buttonAction="QUERY")
    QUERY($tablePtr->)
    $fl_RefreshInfos:=(OK=1)

  : ($buttonAction="PROPAGATE")(We will see this one later...)
    $fl_RefreshInfos:=True

  : ($buttonAction="SUBSET")
    USE SET("UserSet_"+$tableName)
    $fl_RefreshInfos:=True
...
  : ($buttonAction="OK")
    SAVE RECORD($tablePtr->)
```

```
    ar_FieldsInList{0}{$arrayIndex}:=""
    $fl_Redraw:=True

  : ($buttonAction="FIRST")
    FIRST RECORD($tablePtr->)
    $fl_RefreshInfos:=True
...

  Else

End case

If ($fl_RefreshInfos)
  If ($fl_IsPage)
    EXP_DB_PropagateSelection ($tableName;"PAGE")
    EXP_DB_SetButtonsState ($tableName;"PAGE")
  Else
    EXP_DB_PropagateSelection ($tableName;"LIST")
  End if
End if

If ($fl_Redraw)
  $show:=1
  $evt:=-100
  EXP_Explorer_ShowTable (->$show;$tableName;$evt)
End if
```

ステップ5: アクションの結果，サブフォームを更新する必要が発生するかもしれません。たとえば，「最初レコード」ボタンがクリックされた場合，同じボタンバーの「最初レコード」「前レコード」ボタンを disable 状態に変更する必要があります。これを処理しているのが ***EXP_DB_SetButtonsState*** メソッドです。

```
$tableName:=$1
$where:=$2  //LIST or PAGE

$tablePtr:=EXP_Util_GetTablePtrFmName ($tableName)
```

```
$tableNum:=Table($tablePtr)

Case of
  : ($where="LIST")

  Else   //Therefore, it's a Page

    $buttonsName:="_SRC_PAGE_BUTTONS_"
    $buttonsPrefix:="_PAGE_BUTTON_"
    $fl_IsFirstRecord:=(Selected record number($tablePtr->)<=1)
    $fl_IsLastRecord:=(Selected record number($tablePtr->)>=
                              Records in selection($tablePtr->))

    $toolbarName:=Replace string($buttonsName;"_SRC_";"_OBJ_")+
                                                    $tableName+"_"
    $flOK:=False
```

必要な情報が揃ったところで，メソッドをサブフォームのコンテキストで実行します。

```
    EXECUTE METHOD IN SUBFORM($toolbarName;
            "EXP_SubForm_SetButtonsState";$flOK;$buttonsPrefix;
            $fl_IsFirstRecord;$fl_IsLastRecord)

End case
```

ステップ6: コールされている ***EXP_SubForm_SetButtonsState*** メソッドは，渡されたパラメーターに基づき，適切なボタンを enabled または disabled 状態に設定します。

```
$buttonsPrefix:=$1
$fl_IsFirstRecord:=$2
$fl_IsLastRecord:=$3

$ok:=True

$ptr:=OBJECT Get pointer(Object named;$buttonsPrefix+"FIRST_")
If (Not(Nil($ptr)))
  OBJECT SET ENABLED($ptr->;Not($fl_IsFirstRecord))
```

```
End if

$ptr:=OBJECT Get pointer(Object named;$buttonsPrefix+"PREVIOUS_")
If (Not(Nil($ptr)))
  OBJECT SET ENABLED($ptr->;Not($fl_IsFirstRecord))
End if

$ptr:=OBJECT Get pointer(Object named;$buttonsPrefix+"NEXT_")
If (Not(Nil($ptr)))
  OBJECT SET ENABLED($ptr->;Not($fl_IsLastRecord))
End if

$ptr:=OBJECT Get pointer(Object named;$buttonsPrefix+"LAST_")
If (Not(Nil($ptr)))
  OBJECT SET ENABLED($ptr->;Not($fl_IsLastRecord))
End if
```

EXP_DB_PropagateSelection ($tableName;"PAGE") は完全な汎用メソッドです。データベースに定義されたリレーションを解析し，下記のような処理をします。

- テーブル $tableName にセレクションが存在すればリレーションを追跡します。
- リスト形式であれば，セレクション全体が対象となります。
- ページ形式であれば，カレントレコードが対象となります。
- それぞれのリレートテーブルにつき，対応するセレクションを作成します。
- フォームにそのセレクションを表示します。

```
  //First we will create the list of relations 1 to N and N to 1
ARRAY POINTER($ar_1ToN;0)
ARRAY POINTER($ar_NTo1;0)
EXP_DB_FindRelations (->$ar_1ToN;->$ar_NTo1)
  //We will also keep track of the table names where a selection has
been done in order to not do it twice
ARRAY TEXT($ar_TableIsDone;0)
```

```
  //We start from the table that has been clicked last
$tablePtr:=EXP_Util_GetTablePtrFmName ($tableName)

If ($listOrPage="LIST")  //It comes from a list
  //So we should use the current selected lines
  //First, we will Push the current selection in order to switch back
to it
  $setName:="UserSet_"+$tableName
  COPY SET($setName;"temp")

  $selectionName:="TempSel_"+$tableName
  CUT NAMED SELECTION($tablePtr->;$selectionName)

  //Then we will use the current selected records
  USE SET($setName)

  //We call the method to propagate the selection from this table
  $fl_Found:=EXP_DB_PropagateSelection_1Tabl ($tablePtr;->$ar_1ToN;
                                          ->$ar_NTo1;->$ar_TableIsDone)

  //Then we restore the selection
  USE NAMED SELECTION($selectionName)
  //No need to clear it, for we used CUT and not COPY
  //and highlight the previously highlighted records
  COPY SET("temp";$setname)
  //Corresponding to HIGHLIGHT RECORDS($tablePtr->;$setname)

Else   //It comes from a Page
  //So it will be the current record

End if
```

```
  //We can also know which tables can be used for display, in case we
need it
ARRAY TEXT($ar_TablesPresentInForm;0)
EXP_DB_FindPossibleTables (->$ar_TablesPresentInForm)

  //But here, we will look for tables actually used for display
ARRAY TEXT($ar_TablesUsedInLists;0)
ARRAY TEXT($ar_TablesUsedInPages;0)
EXP_DB_FindDisplayedTables (->$ar_TablesPresentInForm;->
$ar_TablesUsedInLists;->$ar_TablesUsedInPages)

For ($i;1;Size of array($ar_TablesUsedInLists))
  EXP_DB_DisplayInfos ($ar_TablesUsedInLists{$i};"LIST")
End for
For ($i;1;Size of array($ar_TablesUsedInPages))
  EXP_DB_DisplayInfos ($ar_TablesUsedInPages{$i};"PAGE")
End for
```

スケーラビリティに向けたプログラミング

v16マルチスレッディングに備える

64ビット版の登場により，4Dが使用できるRAMのサイズに上限がなくなり，事実上，キャッシュやメソッドのスタックに好きなだけメモリが割り当てられるようになりました。4D v15 R5およびv16では，マルチコアおよび/または複数のCPUを搭載したマシンの処理力が存分に活用できるようになり，**スケーラビリティ**は容易に手が届く，現実的なものとなりました。

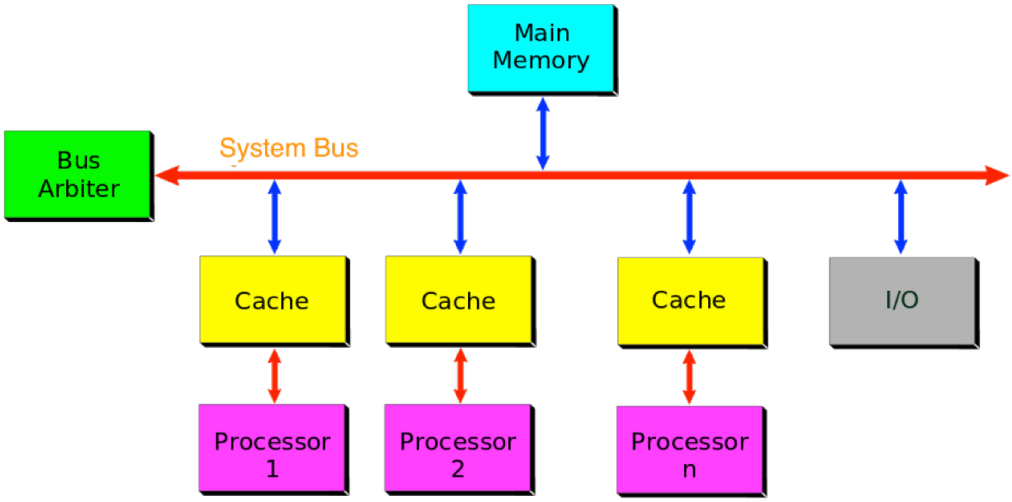
とはいえ，マルチコアCPUのマシンで実行しただけで，勝手にパフォーマンスがアップするわけではありません。マルチスレッディングの恩恵にあずかるためには，その仕組みを正確に理解する必要があります。この章では，4Dアプリケーションのパフォーマンスにおけるこの重要な一歩に対し，どのように備えることができるのかを考慮します。

並列および非同期処理プログラミング

並列プログラミングは，テクニックの一種で，仕事をいくつもの小さな仕事に分割し，別々の処理を同時に実行すれば，全体として早く仕事が完了できる，という原則に基づいています。並列プログラミングはすべての処理に適用できるわけではありません。ある種のアルゴリズムは並列化が不可能ですが，データベース処理の多くは何らかの点で並列化できるものです。

並列プログラミングは，非同期プログラミングと密接な関連があります。アルゴリズムの各パートは，別のパートが完了するまで待機したり，互いに連絡を取り合ったりはしません。言い換えるならば，プロセスに割り当てられる処理は，別のプロセスの状態に依存するようであってはならないということです。

8コアは1コアの8倍も高速であると期待するのは，よくある勘違いです。コンピューターがどんなに多くのコアを積んでいたとしても，システムバスとRAMは1個しかありません。CPUまたはコアのRAMに対するアクセスはいつでもシーケンシャル処理であり，本来，完全に独立しているはずのプロセスも，そこでは待たされることになります。処理の内容がおもに計算であれば，あまり問題にならないこですが，データベース処理の場合，CPUの内部キャッシュよりもずっと多くのメモリが必要になるため，コア数が多ければ多いほど，これは深刻な問題になります。



並列処理でもっとも厄介な問題のひとつは，その同時性に起因しています。ある種の仕事は，独立的に実行することができますが，中には相互に依存しており，したがって同期が取られなければならないものもあります。プロセスが別のバックグラウンドプロセスを起動し，時間を要する処理をそのプロセスに実行させる場合，最初のプロセスは，2番目のプロセスに委ねた処理の結果が必要になるまで，そのまま非同期で実行を続けることができます。

プロセス間の同期を取るために，セマフォが使用できる，と思うかもしれません。2番目のプロセスは計算を開始する前にセマフォをセットし，結果が出ると同時にセマフォを解除します。1番目のプロセスは結果が必要になるところまで処理を続け，そこまで達したらセマフォがクリアされるまで待ち続ける，という仕組みです。しかし，残念ながら，プリエンプティブモードでそれはできません。

マルチプロセスのアプリケーションは，時間を要する処理の多くをバックグラウンドで実行しているため，そうではないアプリケーションよりもスムーズに動作するものです。反面，そのようなアプリケーションは往々にしてデバッグがずっと困難になります。気をつけていないと，デッドロック，つまり2個のプロセスが互いを待ち合っているような状況に陥ってしまいます。しかも，タイミングが肝要なので，その問題はたまにしか発生しません。そして，マーフィーの法則が暗示しているように，デバッグ中にはなぜかその問題が発生しないのです。

憶えておきたいのは次の重要な点です。

スレッドとプロセスは同じものではありません。

4Dデベロッパーは，プロセスのことを互いに干渉することなくメソッドを同時に実行できるもの，として捉えていることが少なくありません。その見方は，間違いではありませんが，実態は少し違います。

4Dの場合，**プロセス**は多くのものを共有しています。インタープロセスと名の付くもの（変数，セット，命名セレクションなど）はすべて共有されているので，4Dのプロセスは別々のスレッドで実行することができません。そこで，4Dのプロセスはスケジューラーによって管理され，自分の番が回ってきたときにだけ実行されるようになっています。

スレッドは，システムレベルの概念であり，オペレーションシステムのスケジューラーによって管理できる，独立した最小単位のプログラム命令の集合と定義することができます。

コオペラティブなマルチタスキング（タイムシェアリングまたは非プリエンプティブマルチタスキング）は，オペレーションシステムが実行中のプロセスから他のプロセスへコンテキストの切り替えをしないタイプのマルチタスク処理です。このシステムでは，実行中のプロセスがある時点で処理を自ら中断し，自主的に制御を他のプロセスに譲ることが関係しています。

プリエンプティブなマルチタスキングは，システムがプロセスの協力を必要とせずに，その処理に割り込んでタスクを中断したり，再開させたりするマルチタスク処理を指します。

アプリケーション内でメソッドを実行するプロセスは，4Dスケジューラーによって管理されており，互いに協力し合わなければなければなりません。同じスレッドで実行されているので，デベロッパーは，特定のプロセスが4Dのランタイムエンジンをコールせずに無限ループを実行するなど，スレッドを独占するようなことがないよう，注意する必要があります。

4Dが自動的に起動するプリエンプティブなプロセスには，1MBのスタックメモリが割り当てられます。このサイズは，SET DATABASE PARAMETER のセレクター53番で変更することができます。この値をおおきくすれば，各クライアントのために作成されるプリエンプティブスレッドの合計が無視できない量になってしまい，同時に接続できるクライアントの数が減るかもしれません。反対に，512KBあるいは256KBに抑えることもできます。ここは最適化のセンスが問われる部分です。変数テーブルのサイズや，メソッド連鎖の数など，メモリを欲するものは，他にもいろいろあるからです。

プリエンプティブ・マルチスレッディング: どのように備えられるか

v15の時点で，4D Server は核となる部分の3/4でプリエンプティブマルチスレッディングを採用していました。つまり，4D Server を構成している主要なサーバー3個のことです。

a. DB4D:

データベースサーバーとも呼ばれ，データそのものに対するアクセス全般を統御するデータベースエンジンがこれに含まれています。他のサーバーは，必ずDB4Dを介してデータにアクセスし，直接，データにアクセスすることはしません。デスクトップ版のDB4Dはコオペラティブ，サーバー版のDB4Dはプリエンプティブです。スレッドは，同じメモリコンテキスト上で動作するプロセスのことである，ということができます。一般にスレッドが軽量とされているのは，その切り替えに際してメモリコンテキストの変更が発生しないためです。

DB4Dの仕事は，基本的にプリエンプティブであり，プリエンプティブスレッドを使用して実行されることになります。クライアント側でローカルではないプロセスが作成されるたびに，サーバー側で対応するプロセスが作成されます。このプロセスはコオペラティブであり，4Dが両プロセス間の同期を取っています。4D Server は，それらに加えてプリエンプティブなプロセスのペアもクライアントプロセス毎に作成します。コオペラティブなスレッドは DELAY PROCESS や GET PROCESS VARIABLE といったアプリケーションのリクエストを処理し，プリエンプティブなスレッドはクエリ，並び替え，インデックスの管理といったデータベースのオペレーションを処理します。これらのオペレーションは，複数のコアに分散することがあります。加えて，Begin SQL/End SQL ブロック内に記述されたSQLリクエストを処理するためのプリエンプティブスレッドが作成されることもあります。

b. アプリケーションサーバー

4Dプロセスを実行し，メソッドを評価するためのサーバーです。4D Remote からのリクエストを処理し，必要な数の4Dプロセスを作成し，データを処理したりクエリを実行したりする場合には，DB4Dと対話します。

c. SQLサーバー

SQLクエリを処理するためのサーバーです。SQLクエリを解析し，処理するためのSQLエンジンが含まれますが，4Dコールに変換したSQLクエリがDB4Dに渡されているのであり，SQLサーバーが直接データにアクセスするわけではありません。4D v11 SQLで初めて登場したときから，このサーバーはマルチスレッディングに対応したアプリケーションであり，マルチコアアーキテクチャの恩恵にあずかることができました。デベロッパーは，オペレーションシステムがスレッドに処理を分散する様子を観察し，コードを書き換えることによって，データベースの速度を向上させることができます。たとえば，CPUサイクルをチェックすることにより，ほんとうにマルチコアモデルが活用されているか，ある程度は判断することができます。

データアクセス処理，インデックスおよびデータベースの管理全般，さらにSQLエンジンのコールは，プリエンプティブマルチスレッディングに対応しており，コア数を有効に利用することができます。これには，キャッシュのフラッシュ，インデックスの作成，ログファイルの作成，といった操作が含まれます。

d. HTTPサーバー

HTTPクエリおよびREST (**Re**presentational **S**tate **T**ransfer) リクエストを処理するためのサーバーです。HTTPおよびRESTの原則を踏襲したAPI，と言い換えても良いでしょう。普通はWebサーバーとして運用されますが，RESTやJSONといったテクノロジーを使用したデータサーバーとしても使用することができます。HTTPサーバーが直接データにアクセスすることはなく，DB4Dと対話することにより，データを処理したりクエリを実行したりします。

このように，4Dは多数のプロセスを同時に実行しているのであり，マルチタスクであるということができます。あるプロセスがユーザーからの入力を待っている間，別のスレッドは実際にクエリを実行しているかもしれません。このマルチタスキングはシステムスレッドを使用しているため，マルチコアあるいはマルチプロセッサ搭載マシンの能力を存分に引き出すことができます。結果的に全体のパフォーマンスにも向上と安定が期待できます。

多数のクライアントが同時に接続し，同時にさまざまなデータベースオペレーションを実行できるということは，スケーラビリティにも寄与します。それらのオペレーションは別々のプリエンプティブスレッドで実行されるためです。この並行処理モデルのメリットは，CPUの数に比例してアプリケーションの規模も拡大できる点にあります。それぞれのクライアントプロセスは，処理能力の向上から均等に益を受けます。

インデックスの管理，データキャッシュ，SQLサーバーに対する外部コールの処理は，それぞれ定められたプリエンプティブスレッドで実行されます。こうした処理のために，プロセスの数だけプリエンプティブスレッドを作成して管理する必要はありません。いずれにしても，プリエンプティブスレッドの制御は基本的にオペレーションシステムの仕事となります。

v15以前，ほんとうの意味でマルチスレッディングの恩恵にあずかることができたのは，4D Server のほうであり，シングル版で得られるメリットは限定的でした。v16では，アプリケーションサーバーもプリエンプティブマルチスレッディングが使用できるようになります。

ワーカー，プロセス，ウインドウコンテキストを理解する

v15R5およびv16以降，**ワーカー**と呼ばれる新しいタイプのプロセスが存在します。ワーカーは基本的には通常のプロセスと同じですが，両者間には以下のような違いがあります。

- プロセスは **New process** で開始します。

```
$process:=New process("MyMethod";$stackSize;"Name";$param1;$param2;…)
```

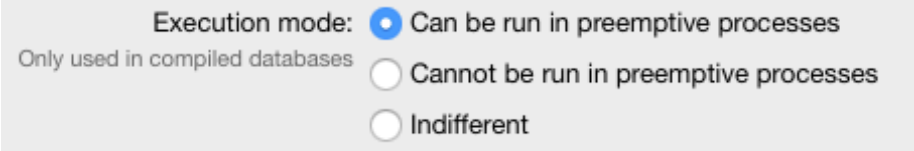
- コマンドはプロセス番号を返し，それを介してプロセスにアクセスすることができます。メソッド MyMethod はパラメーターを受け取り，最後までコードを実行し，その後，4Dによってプロセスは終了させられます。

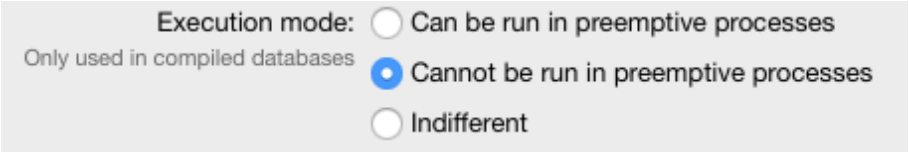
- ワーカーは **CALL WORKER** で開始します。

```
CALL WORKER("WorkerName";"MyMethod";$param1;$param2;…)
```

- ワーカーを識別するのはその名前であり，番号ではありません。メソッド MyMethod はパラメーターを受け取り，最後までコードを実行しますが，その後，いつでも新しいメソッドを実行できる状態のままメモリに残り続けます。

- ワーカーあるいはプロセスがコオペラティブまたはプリエンプティブというわけではありません。そうではなく，どちらもコオペラティブまたはプリエンプティブスレッドによって実行することができ，それを決めるための新しい設定 "実行モード" がメソッドプロパティに追加されました。第1の選択肢はプリエンプティブモード，第2の選択肢はコオペラティブモードでメソッドが実行されることを意味します。





- プロセスとは，そのプロセスがフォームを表示していれば CALL PROCESS コマンド，そうでなければ，外部から SET PROCESS VARIABLE や VARIABLE TO VARIABLE のようなコマンドで書き換えられた変数をそのプロセス内で参照することによって通信します。

- ワーカーとは，**CALL WORKER** コマンドで通信します。各ワーカーは "メールボックス" を持っており，ワーカーをコールすると，そのメールボックスにメッセージが投函されます。実行中のメソッドを終了したワーカーは，メールボックスの中を確認し，メッセージをみつけると，それを処理します。コールと同時にメッセージが処理されるわけではありません。つまり，この処理は非同期です。

- ワーカーとプロセスは，どちらもプリエンプティブモードでメソッドを実行できるとはいえ，できればワーカーを使用し，ほんとうに必要がなくなるまではメモリに残しておくことが強く勧められています。プロセスまたはワーカーを作成することは，4Dにとってかなりの負担になるためです。

- プロセスは，メソッドの最後まで到達するとすぐに終了しますが，ワーカーはメソッド終了後も別のメソッドを実行できる状態のまま存在し続けています。終了させるためには，**KILL WORKER** コマンドを明示的に使用する必要があります。

- コオペラティブモードであれば，プロセスまたはワーカーはどんなメソッドでも実行することができます。一方，プリエンプティブモードでは，実行できるメソッドに次のような条件があります。

    - ユーザーインタフェースがないこと
    - インタープロセス変数を使用していないこと
    - コンパイルモードで実行していること
    - SET/GET PROCESS VARIABLE, VARIABLE TO VARIABLE を使用していないこと

- フォームを表示しているプロセスとは，**CALL FORM** で通信することもできます。

```
CALL FORM($window2call;"MyMethod";$param1;$param2;…)
```

この新しいコマンドは，フォームをコールしているというよりは，ワーカーと同じメッセージングの仕組み（メールボックス）を使用し，ウインドウとの通信を実現するものです。それぞれのウインドウは，メールボックスを持っています。上記コマンドは， `$window2call` で指定されたウインドウに関連づけられたコンテキストをコールし，$param1, $param2 といったパラメーターを渡してメソッド MyMethod を実行させる働きをします。任意のプロジェクトメソッドを実行することができ，そのウインドウに表示されているフォームオブジェクトにアクセスすることができます。

メッセージング，スライシング，チャンキング

ここまで説明してきたようなスケーラビリティを実現するためには，**新しいスタイルのプログラミング**を実践しなければなりません。コードを組み立てる上での考え方をどのように変えなければならないのかを示すために，"testWorkerJPR" というサンプルを用意しました。

メッセージング

サンプルに収められているメソッドは，そのほとんどがJSONオブジェクトをパラメーターとして使用し，そのオブジェクトを特定のプロセスに送ることにより，プロセス間の通信を成し遂げています。つまり，従来とは異なる方法でメッセージングを実現しています。従来のプログラミングでは，（<>messageのような）インタープロセス変数を外部からセットすることにより，別プロセスをコールしていました。

```
<>Message:="GRAPH"
CALL PROCESS(1)
```

コールされたほうのプロセスは，次のようにメッセージを処理しました。

```
Case of
  : ($event=On Outside Call)
    $message:=<>Message
    <>Message:="GRAPH"
    Case of
      : ($message="INIT")

      : ($message="GRAPH")
        //Code is here
```

新しいスタイルでは，メッセージングの仕組みそのものをコーディングすることはしません。そこは4Dが自動的に処理してくれるからです。

```
OB SET($object;"WHAT2DO";"IDENT_PROCESS")
OB SET($object;"THERMOID";$thermoID)
OB SET($object;"WORKERMODE";$mode)
CALL FORM($window2call;"CF_Talk2Form";$object)
```

参照 $window2call のウインドウに表示されているフォームがコールされ，プロジェクトメソッド CF_Talk2Form が実行されます。

```
$object:=$1
$what2do:=OB Get($object;"WHAT2DO")
$mode:=OB Get($object;"WORKERMODE")
$thermoID:=OB Get($object;"THERMOID")
Case of
  : ($what2do="IDENT_PROCESS")
```

メソッド CF_Talk2Form はウインドウに表示されている**フォームのコンテキスト**で評価されます。つまり，そのフォームに表示されているフォームオブジェクトに変数名（ここでは ar_ProcessNb）またはオブジェクト名（"_THERMO_00"）でアクセスすることができます。

```
ar_ProcessNb{$thermoID}:=OB Get($object;"CURPROCESS")
OBJECT Get pointer(Object named;"_THERMO_00")->:=$percent
```

メッセージングの仕組みにより，ウインドウに表示されたフォームだけでなく，ワーカーもパラメーターを渡してコールすることができます（**CALL WORKER**）。ただ，ここで紹介しているのはいずれも**非同期プログラミング**なので，コールと同時にメソッドが実行されると考えてはいけません。コールされたワーカーがメソッドを実行するのは，ワーカーが実行中のメソッドを終え，メッセージボックスから新しいメッセージを取り出した後のことになります。

スライシング

ある程度の時間を要する仕事や，複雑な仕事を分割し，それぞれが互いに独立したプリエンプティブなプロセスによって別々に処理できるよう振り分けることを**スライシング**と言います。サンプルデータベースには，そのような例が収められています。

```
$nbRecords:=Records in table([Cities])
If ($count=0)
  $sliceSize:=$nbRecords
Else
  $sliceSize:=$nbRecords\$count
End if

OBJECT SET VISIBLE(*;"_TIMER_";True)
```

```
$sliceNb:=0
For ($i;1;Size of array(ar_ProcessNb))
  If (ar_ProcessNb{$i}>0)
    $sliceNb:=$sliceNb+1
    $first:=(($sliceNb-1)*$sliceSize)+1
    $last:=$first+$sliceSize
    If ($last>$nbRecords)
      $last:=$nbRecords
    End if
```

タスクの内容は，150万件のレコードから統計的なレポートを作成するというものです。ワーカー数は，変数 $count に代入されています。各ワーカーに割り当てられる仕事の量が計算され，スライスのサイズ $sliceSize が決まります。$first と $last は，各ワーカーが処理するべきレコードの範囲です。たとえば，8個のワーカーを使用するのであれば，それぞれに仕事の1/8を割り当てることになります。しかし，8コアCPUのコンピューター上で実行したとしても，処理速度が8倍になるわけではありません。レコード，つまりアドレステーブルに対するアクセスが発生するときには，内部的な排他ロック（ミューテックス）が使用されるためです。それでも，全体の処理はずっと高速になることでしょう。コア数以上に仕事を細かく分割してはいけません。1個のコアに1個のタスクを割り当てることに意味があります。

すべてのアルゴリズムがスライスできるわけではないでしょう。あるいは，別の方法でタスクを分割する必要があるかもしれません。

チャンキング

複数のワーカーが効率的に仕事を進められるようにするためのテクニックのひとつにチャンキングがあります。たとえば，いくらか時間を要する処理（数分間）にワーカーが取りかかっている状況を想像してください。この処理を中断するためには，ワーカーをコールし，仕事の途中であっても処理を終了するように命令しなければなりません。サンプルでは，下記のようなコードが使用されています。

```
: ($what2Do="BANG")
    OB SET($object;"WHAT2DO";"AGONY")
    CALL FORM($window2call;"CF_Talk2Form";$object)
    KILL WORKER
```

しかし，ワーカーがメッセージを受け取るのは，実行中のメソッドを終了した後であり，それはずっと後のことになるかもしれません。そうではなく，実行中のワーカーにも外部からメッセージを送ることができるようにするためには，ワーカーの処理を時間のチャンクに分割し，定期的（たとえば2分毎）に必ずメッセージを確認して，作業を中断できるようなポイントを設けることが必要です。たとえば，次のようなことができます。

- 計算途中の状態や値など，使用中のローカル変数を1個のオブジェクト変数にコピーする
- そのオブジェクト変数をパラメーターとしてワーカー自身にメッセージを送る
- メッセージを受信したらローカル変数を再現し，中断したところから処理を再開する

あるいは，オブジェクトの代わりにプロセス変数を使用しても構いません。ワーカーを終了するまで，プロセス変数は残されています。

こうした理由により， For...End ループよりも Repeat...Until ループをのほうがワーカーのコーディングには適しています。前者の場合，ループを途中で中断しなければなりませんが，それはあまり推奨されていない方法です。