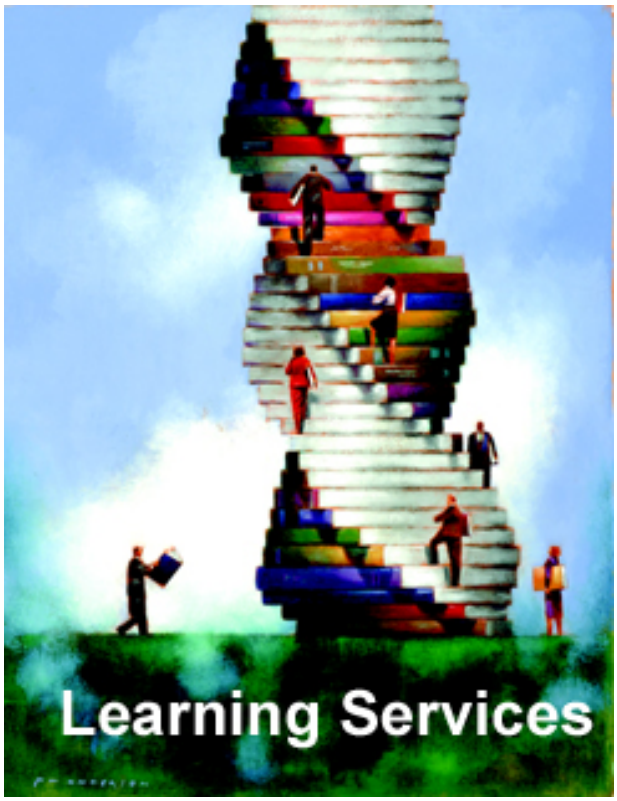


# 多次元データエンジンの

# 概念およびアーキテクチャ

（IRIS Version 2024.1 ベース）

V1.0

2024 年 12 月

インターシステムズジャパン株式会社

## 1. はじめに

この文書は、IRIS のデータベースストレージエンジンである多次元データエンジンについてその概念、および大まかなアーキテクチャを述べたものです。
概念、アーキテクチャを理解することで、アプリケーション開発において、IRIS の能力を十分引き出す上でのヒントになれば、幸いです。

## 2. 多次元データエンジン概要

多次元データエンジンを一言で表現することは難しいですが、IRIS のデータベースのコアである多次元データ構造にアクセスするための作法、データ管理機構を総称してそう呼んでおり、そのアクセスを実現するために ObjectScript 言語にそのシンタックス、セマンティクスが定義されています。

## 3. ObjectScript

IRIS は、データベース管理システムであると共に、開発言語を含む開発環境も提供します。
現在では、ほとんどの商用およびオープンソースのリレーショナルデータベースシステムにもストアドプロシジャを記述するためのスクリプト言語が装備されていますが、それらは、一連の連続する処理を一度に実行することを主眼に設計されており、本格的なビジネスロジックを記述することを想定していません。
一方、IRIS のスクリプト言語は、ビジネスロジックを記述することを前提として設計された言語であり、一般的なプログラミング言語（C、Java、Javascript,Python など）に遜色のない記述能力を提供し、かつデータベースを使ったアプリケーション開発という観点からは、他に比類のない生産性を提供します。

## 4. 多次元配列変数と永続変数

ObjectScript には一般的なプログラミング言語にはない変数に関する 2 つの非常にユニークな特徴があります。

プログラミング言語に欠かせない要素として、変数というものがありますが、その変数にもスカラ変数と配列変数の 2 種類があるのが一般的です。
配列変数は、次元要素として正の整数を指定することができ、たとえば、A(10,100)という表現形式で、1 次元目に 10 のマス、2 次元目に 100 のマスを用意し、合計 10*100 の 1000 個のマスを割り当て、その個々のマスに値を割り当て、あるいは参照していくという処理をプログラミング言語で記述していきます。

一方、ObjectScript では、一般的な配列変数と異なり、その配列の要素として、正の整数だけでなく、可変長の文字列を指定することができます。
この特徴により、一般的な配列変数よりもずっとデータの表現力が高まります。

もうひとつの特徴として、変数の属性として永続性という概念をサポートしている点があげられます。
プログラミング言語の変数は、一般的にプログラムの実行中に存在し、そのプログラムが終了すると消失します。
永続性のある変数という概念により、プログラム終了後にもその変数構造を永続性のある媒体、通常はハードディスクや SSD に保持することができます。
これにより、あとからそのデータを参照、更新、あるいは削除を行うことができます。
さらに複数のプロセスから同時にアクセスすることが可能です。

## 4.1. 多次元配列変数によるデータモデル表現

多次元配列変数を使って、データモデルを表現する例を見てみましょう。

### 4.1.1. モデル概要

衣料販売店の受発注、在庫管理のシステムを考える時の基本情報としての衣料品情報についてモデル化してみます。

| | |
|---|---|
| 商品基本情報 | 商品番号を主キーとし、それに従属するデータを管理<br>従属データ： 商品名、製造会社 ID など |
| 商品価格情報 | 商品番号+サイズを主キー（2 つのフィールドの複合キー）とし、それに従属するデータを管理<br>従属データ： 価格 |
| 商品在庫情報 | 商品番号+サイズ+色を主キー（3 つのフィールドの複合キー）とし、それに従属するデータを管理<br>従属データ: 在庫数 |

**表 1 衣料販売店：衣料品情報**

ここで、リレーショナルテーブルでの設計に慣れている方は、色やサイズなどを直接キーとして持つのではなく、別テーブル（色テーブル、サイズテーブル）を設計し、商品価格情報や商品在庫には、そのキーを持つという設計を考えると思います。
もちろん、その様な設計も可能ですが、ここではモデルを少し簡略化して考えてみます。

### 4.1.2. 多次元配列変数表記法

上記のモデルで多次元配列変数を使って、表現してみましょう。

```
PRODUCT("A00001") = "ポロシャツ^12"          商品基本情報
PRODUCT("A00001","S") = 3200                 商品価格情報
PRODUCT("A00001","M") = 3800                 同上
PRODUCT("A00001","L") = 4000                 同上
PRODUCT("A00001","XL") = 4200                同上
PRODUCT("A00001","S","青") = 10              商品在庫情報
PRODUCT("A00001","S","赤") = 5               同上
PRODUCT("A00001","S","オレンジ") = 20 同上
PRODUCT("A00001","M","青") = 18              同上
PRODUCT("A00001","M","赤") = 15              同上
PRODUCT("A00001","M","オレンジ") = 10同上
```

上記で PRODUCT は、変数名を表します。
配列の 1 次元目の要素に"A00001"というのがあり、これが商品番号を表します。文字列表現は、"（ダブルクォート）で囲みます。各次元の要素のことをサブスクリプトあるいは添え字と呼びます。
2 次元目の"S","M"などは、サイズを表します。
3 次元目の"青","オレンジ"などは、色を表しています。
上記の 1 つ 1 つの変数表現を変数ノードと呼び、変数ノード毎にデータを格納する箱があるというイメージになります。
各変数ノードの箱には、最大約 36MB までの文字列を格納することができます。
上記の例では、PRODUCT("A00001") = ポロシャツ^12 という表現は、ポロシャツ^12 という文字列が格納されていることを表していますが、意味的には、その文字列は、^で区切られた 2 つのフィールド（カラム）で構成されていることを表しています。
テーブルでの複数フィールドで構成されるレコードと同等の構造を表現するためには、このように区切り文字を使う方法と、後ほど説明する LIST 構造を使用する方法があります。
テーブルの場合、各カラム別にデータのタイプを指定するのが、一般的ですが、ObjectScript 変数は全てタイプレス（全てバイト列）です。
但し、全てがバイト列だからといって、数字や日付、バイナリデータを処理できないという意味ではありません。

### 4.1.3. 永続変数表記法

[4.1.2 多次元配列変数表記法] では多次元配列変数の表記法について着目しましたが、それでは、永続性についてどう表現するのか見てみましょう。

永続性があるかないかを区別する方法は、変数名の先頭に^（山形記号）をつけることで表現しています。
つまり PRODUCT は、永続性のない変数、^PRODUCT は、永続性のある変数となります。
このことは、永続性のあるなしにかかわらず同じデータ表現、つまり同じデータの構造を表現する能力を有するということを意味します。
この^のついた変数をグローバル変数と呼びます。
（他のプログラミング言語でのグローバル変数、つまり大域変数と概念が違っている点、注意して下さい。）

### 4.1.4. 多次元配列変数で表現できるデータモデル鳥瞰図

下の図は、多次元配列変数で表現できる以下のデータモデルを鳥瞰的に見たものです。

^PRODUCT({商品 ID}) = {商品名}^{製造会社 ID}
^PRODUCT({商品 ID}, {サイズ}) = {定価}

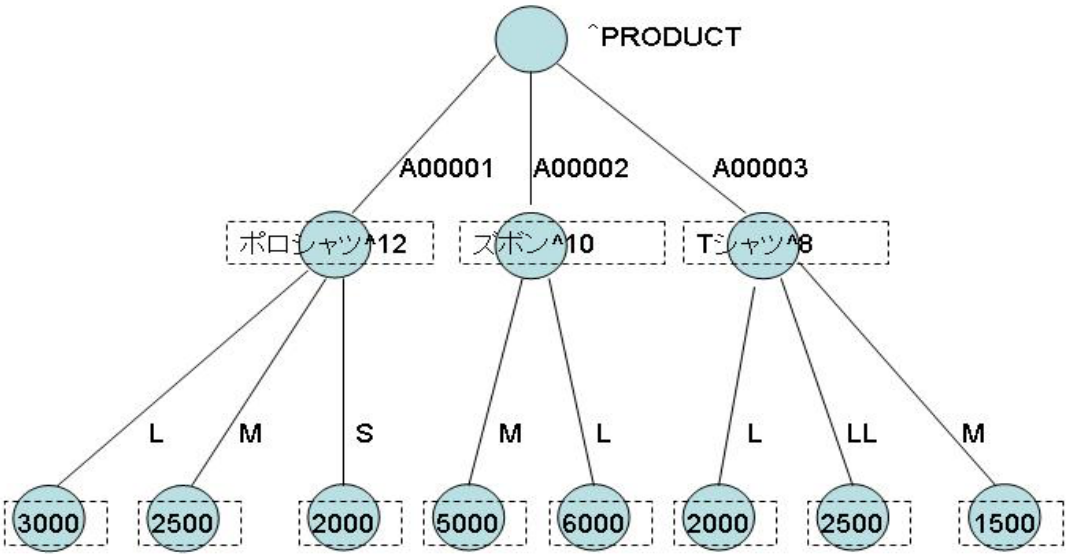


**図 1 データモデル鳥瞰図（^PRODUCT）**

このモデルは、複数のキーから値を特定すると言う意味では多次元ですが、キーによるデータの検索と言う観点から言えば、第 1 キー、第 2 キー…という順にアクセスするのが最も効率的であるので、上記のツリー構造のモデルとして表すほうが、より実際の検索効率を表していると言えます。

# 5. 多次元配列変数操作法基礎

この章では、多次元配列変数をプログラムからどのように操作するか見ていきましょう。
まず変数操作の基本である変数の参照および変数に値を割り当てる方法から見ていきましょう。

## 5.1. 参照

変数への参照は、一般的なプログラミング言語と同様、変数（ノード）名を指定することで実現されます。

以下は、PRODUCT の変数ノードのひとつの内容を表示する命令実行を表しています。

```
Write PRODUCT("A00001","S","オレンジ")
```

PRODUCT("A00001","S","オレンジ")の内容、つまりさきほどの例（表 1 衣料販売店：衣料品情報）では、20 と返ってきます。

さらに、各添え字の部分に変数を指定することができます。
たとえば、

```
Read "商品番号? ",ID
Read "サイズ? ",Size
Read "色? ",Color
Write PRODUCT(ID,Size,Color)
```

という 4 行のプログラムを実行し、商品番号、サイズ、色をユーザが入力すると、
ID,Size,Color という変数にそれぞれ入力した値が入ります。
商品番号に A0001、サイズに S、色にオレンジと入力すると、

PRODUCT(ID,Size,Color)という表現は、実行時に PRODUCT("A00001","S","オレンジ")
と置き換えられて解釈され、最終的に PRODUCT("A00001","S","オレンジ")の内容を
表示して終わります。

## 5.2. 値の割り当て

変数への値の割り当てには、Set コマンドを使用します。

例えば、新しい色、紫を追加する場合、

```
Set PRODUCT("A00001","S","紫") = 21
```

と記述します。

先ほどと同じように、添字の部分に変数を使用して表現することもできます。

```
Set ID = "A00001"
Set Size = "S"
Set Color = "紫"
Set StockNumber = 21
Set PRODUCT(ID,Size,Color) = StockNumber
```

とすることもできます。

## 5.3. 多次元配列変数の宣言

一般的な言語での配列変数の使用に際しては、予め領域を宣言する必要があります。
多次元配列変数では、領域の宣言はどうするのでしょう？
実は、領域の宣言は必要ありません。
逆にいうと、宣言のしようがないといったほうが、正確です。
配列の要素が、正の整数に限られる場合には、準備する領域は、その整数個分用意すれば、
良いですが、多次元配列変数のように配列の添字に可変長の文字列の可能性があると、
予めどういう領域を確保しておけばよいかを決めることができません。
従って、多次元配列変数ノードは、それに対して、値を初めて割り当てる時に、領域が
確保されます。

## 5.4. スパース（まばらな）配列

多次元配列変数の宣言の説明から言えることは、多次元配列を構成する配列要素は、論理空間としては無限大です。
しかし物理的には存在する（明示的に値を割り当てた）配列ノードしかありません。
この状態をスパース（まばら）な配列といいます。
物理的には、必要なものだけが、コンパクトにぎゅっと詰められて格納されています。
決して、存在しないデータのために領域が取られることはありません。

## 5.5. 多次元配列ノードの削除

それでは、必要なくなったデータは、どうしたらよいでしょうか？
そのまま放っておくということも考えられますが、ObjectScript では、多次元配列変数の理念にのっとり、
必要なくなったものは、やはり削除できたほうが良いという考えを取り入れています。

データの生成も動的、データの削除も動的であるということで整合性がとられています。

削除を行うために Kill コマンドが用意されています。

以下は、先ほど作成した PRODUCT("A00001","S","紫")というノードを削除する例です。

```
Set ID = "A00001"
Set Size = "S"
Set Color = "紫"
Set StockNumber = 21
Kill PRODUCT(ID,Size,Color)
```

さらに Kill コマンドは、上位ノードを消すと、それにつながる下位ノードも削除する性質をもっています。（親を消すと子も消える）

例えば、

```
Kill PRODUCT("A00001","S")
```

を実行すると、先ほどの例では、以下の関連するデータ（2 次元目までのキーが一致するノード）を一度に削除することができます。

```
PRODUCT("A00001","S") = 3200                商品価格情報
PRODUCT("A00001","S","青") = 10             商品在庫情報
PRODUCT("A00001","S","赤") = 5                      同上
PRODUCT("A00001","S","オレンジ") = 20       同上
```

さらに

```
Kill PRODUCT
```

を実行すると、全ての PRODUCT 多次元配列ノードが一瞬のうちに削除されます。
不思議なことに、どんなに大量のノードが配下にあっても、この削除は、一瞬で行われます。

## 5.6. 多次元配列変数ノードの走査 1

ある多次元配列変数のノードが大量にあったとしても、特定のノードへのアクセスは、非常に高速です。
多次元配列変数ノードにアクセスするために変数名とノードを特定するための添字を指定します。
これは、キーを与えることによって、データにランダムアクセスをすることと同等です。

一方、アプリケーションのアクセスパターンとして、ある特定のキーに対応するデータを取得後(ランダムアクセス)、そのキーを基点として連続的にアクセス（シーケンシャルアクセス）することもよく行われます。
これを実現するために ObjectScript には、$Order 関数というものが用意されています。
$Order 関数は、指定された多次元配列変数ノードと同じ次元レベルの次のノードを返します。

例えば、商品番号 A00001 でサイズ S に対して、どんな色のノードがあるかを走査したいという処理を実現するには、以下のような処理を行います。

```
Set color=""
For  {
    Set color=$Order(PRODUCT("A00001","S",color))
    If color="" Quit
    Write "色 : ",color
    Write "  データ : ",PRODUCT("A00001","S",color),!
}
```

上記の例では、3 次元目の色の基点として""(空文字)を指定します。
次に For ループの中で、$Order を使用して順番に 3 次元目の color に何があるかを
見ていきます。
$Order の結果、color に""が返ってくると、走査が終了したことを判定することができます。

## 5.7. 多次元配列変数ノードの走査 2

$Order 関数は、同一次元レベルでの添え字の走査を行うものですが、これを行うためには
ノードの構造を予め理解して実行する必要があります。
一方、どんな（次元）構造のノードがあるかわからずにとにかく順番に走査したいというケースも少なからず
あります。

これを行うためには、$Query 関数が用意されています。

以下は、$Query 関数を使用して、PRODUCT の全ノードを走査するコード例です。

```
Set pnode="PRODUCT"
SET pnode=$QUERY(@pnode@(""))
WRITE !,@pnode
FOR    {
         SET pnode=$QUERY(@pnode)
         if pnode="" QUIT
         WRITE !,@pnode
}
WRITE !,"Finished."
QUIT
```

変数の前の@は、間接表現といって、これも ObjectScript の特有の処理方法ですが、
文字とおり変数の内容を（間接的に）実行するものです。

## 5.8. 存在チェック

多次元配列変数は、動的な構造で、値を割り当てることによって初めて領域が確保され、変数ノードが生成されるという話をしましたが、それでは、存在しない変数ノードを参照したら、どうなるでしょう？

例えば、いままで定義していない以下のデータの参照を行うと、どうなるでしょうか？

```
Write PRODUCT("A00001","S","ネイビーブルー")
```

コマンドモードで上のコマンドを入力すると、

<UNDEFINED>というエラーが表示されます。

これは、その変数ノードが未定義であるというエラーを表示していることになります。

一連のプログラムの中で、未定義データの参照が、必ずエラー終了するというのは、不都合です。
そのため ObjectScript には、変数ノードが定義済みか否かを調べる$Data 関数が用意されています。
$Data 関数は、4 つの値を持ちえます。詳しくは ObjectScript リファレンスを参照して下さい。

なお、未定義変数の参照でエラーを返すかわりに、未定義時の置換文字を返す$Get 関数もあります。
この関数を使用すると、$Data 関数の使用を省略できるケースもあります。

## 5.9. LIST 構造

多次元配列変数のノード毎にデータの箱ができて、そこに値を入れていくという説明をしました。
そこには、最大約 36MB までの文字列が入って、区切り文字を使うことによって、その文字列は、複数のフィールド（カラム）を持つレコードを表現できるという話をしました。
この手法には、2 つの制限事項があります。

1) データとして区切り文字を持つことはできない。

2) データ長が長く、区切り文字で区切られるフィールドの数が多くなると、個々のフィールド値の取得に時間がかかるようになる。

上記の制限事項に対応するため ObjectScript には、LIST 構造というものが用意されており、現在では、区切り文字形式よりは、この LIST 構造を使用することが推奨されています。

LIST 構造を作成するために、$LISTBUILD 関数が用意されています。
LIST の各要素を取得するには、$LIST 関数を使用します。

```
Set list = $ListBuild(12345,"あいうえお",23.456)
Write $List(list,1)
Write $List(list,2)
Write $List(list,3)
```

上記は、3 つのフィールドで構成される LIST を作成し、1 つ 1 つのフィールドを表示する処理を行っています。

## 5.10.多次元配列変数操作演習 1

それでは、簡単な多次元配列変数操作を実際に行ってみましょう。
ここでは、永続変数を使用します。

まず、IRIS ターミナルを起動します。（キューブからターミナルをクリック）
起動後、以下のコマンドを入力し、Enter キーを押します。

```
USER>for i=1:1:100000 set ^A(i)=$lb(i,"abcdef",34.55)
```

これは、3 つのフィールドよりなる 10 万件のレコードを作成したことと同等です。
本当に 10 万件作成できたか不安になるくらいのスピードですよね？

次に、いくつか、レコードの中身を見てみましょう。

```
USER>write ^A(1)

USER>write ^A(1000)

USER>write ^A(100000)
```

いずれのコマンドも一瞬のうちに答えを返すことでしょう。（中身は$LIST 構造なので、文字化けしていますが）

もう 1 つターミナルを起動してみてください。
同様に

```
USER>write ^A(1)

USER>write ^A(1000)

USER>write ^A(100000)
```

というコマンドを発行すると、先ほどのターミナルと同じ結果が返されることがわかると思います。
つまり、これは、プロセス間で多次元変数データが共有されていることを示しています。

次に以下のコマンドを実行します。
これは、先ほど作った 10 万件のレコードを一瞬にして削除する命令です。

```
USER>Kill ^A
```

このコマンドも、一瞬のうちに終わります。本当に全部削除されたのか不安になりますよね。
即座にもう 1 つのターミナルに移って、以下のコマンドを発行してください。

```
USER>write ^A(1)
```

<UNDEFINED>というエラーになると思います。

```
USER>write ^A(100000)
```

これも<UNDEFINED>というエラーになると思います。

本当に 10 万件のデータが消えているのです。

## 5.11.多次元配列変数操作演習 2

ある特定の多次元配列変数の値を各次元の添字を与えることにより高速にアクセスできるようにするためには、変数がある一定の基準で並べられていることが不可欠です。

そこで、多次元配列変数がどのように並べられるのか見てみたいと思います。
ターミナルで、以下の一連のコマンドを実行します。

```
Set ^PRODUCT("A0001")="ポロシャツ"
Set ^PRODUCT("A0001","SS")=2000
Set ^PRODUCT("A0001","SS","赤")=10
Set ^PRODUCT("A0001","SS","茶")=10
Set ^PRODUCT("A0001","M")=3000
Set ^PRODUCT("A0001","M","赤")=10
Set ^PRODUCT("A0001","M","青")=5
Set ^PRODUCT("A0001","L")=3500
Set ^PRODUCT("A0001","L","赤")=7
Set ^PRODUCT("A0001","L","青")=20
Set ^PRODUCT("A0001","L","オレンジ")=3
Set ^PRODUCT("A0001","LL")=3800
Set ^PRODUCT("A0001","LL","赤")=5
Set ^PRODUCT("A0001","LL","青")=9
Set ^PRODUCT("A0001","LL","白")=3
```

作成された変数ノードを走査するには、走査用のユーティリティを使います。

```
USER>do ^%G

Device:
Right margin: 80 =>
Screen size for paging (0=nopaging)? 24 =>
For help on global specifications DO HELP^%G
Global ^PRODUCT
^PRODUCT("A0001")="ポロシャツ"
^PRODUCT("A0001","L")=3500
^PRODUCT("A0001","L","オレンジ")=3
                                "赤")=7
                                "青")=20
^PRODUCT("A0001","LL")=3800
^PRODUCT("A0001","LL","白")=3
                               "赤")=5
                               "青")=9
^PRODUCT("A0001","M")=3000
^PRODUCT("A0001","M","赤")=10
                              "青")=5
^PRODUCT("A0001","SS")=2000
^PRODUCT("A0001","SS","茶")=10
                               "赤")=10

Global ^
```

**図 2 変数ノード走査用ユーティリティ：%G**

実際に表示されたとおりの順番で変数ノードは格納されます。
設定された順番に格納されるのではないことがわかると思います。

並びの順番（照合順といいます）は、

上位ノードが先
同一ノード間では、数字、文字コード順（既定値は、Unicode）の並びとなります。

データを追加してみます。

```
USER>Set ^PRODUCT("A0001","M","水色")=4
```

再度走査してみると、上で設定された変数ノードが追加されているのが確認できると思います。

```
USER>do ^%G

For help on global specifications DO HELP^%G
Global ^PRODUCT
^PRODUCT("A0001")="ポロシャツ"
^PRODUCT("A0001","L")=3500
^PRODUCT("A0001","L","オレンジ")=3
                              "赤")=7
                              "青")=20
^PRODUCT("A0001","LL")=3800
^PRODUCT("A0001","LL","白")=3
                            "赤")=5
                            "青")=9
^PRODUCT("A0001","M")=3000
^PRODUCT("A0001","M","水色")=4
                            "赤")=10
                            "青")=5
^PRODUCT("A0001","SS")=2000
^PRODUCT("A0001","SS","茶")=10
                            "赤")=10

Global ^
```

今度は、一部のデータを削除してみましょう。
以下のコマンドを実行します。

USER>Kill ^PRODUCT("A0001","LL")

LL 以下のデータがきれいに消えているのがわかります。

```
USER>do ^%G

For help on global specifications DO HELP^%G
Global ^PRODUCT
^PRODUCT("A0001")="ポロシャツ"
^PRODUCT("A0001","L")=3500
^PRODUCT("A0001","L","オレンジ")=3
                              "赤")=7
                              "青")=20
^PRODUCT("A0001","M")=3000
^PRODUCT("A0001","M","水色")=4
                             "赤")=10
                             "青")=5
^PRODUCT("A0001","SS")=2000
^PRODUCT("A0001","SS","茶")=10
                             "赤")=10

Global ^
```

動的な構造というのが、なんとなく理解できましたか？

この動的な構造では、次元の追加も自由自在なのです。

以下のコマンドを実行してみて下さい。

USER>Set ^PRODUCT("A0001","XL","水色")=4

2 次元目の XL はもともと定義されていなかったにもかかわらず、3 次元目の水色である
ノードが追加されているのが、わかると思います。

```
USER>do ^%G

For help on global specifications DO HELP^%G
Global ^PRODUCT
^PRODUCT("A0001")="ポロシャツ"
^PRODUCT("A0001","L")=3500
^PRODUCT("A0001","L","オレンジ")=3
                                "赤")=7
                                "青")=20
^PRODUCT("A0001","M")=3000
^PRODUCT("A0001","M","水色")=4
                              "赤")=10
                              "青")=5
^PRODUCT("A0001","SS")=2000
^PRODUCT("A0001","SS","茶")=10
                              "赤")=10
^PRODUCT("A0001","XL","水色")=4

Global ^
```

動的に構造が生成され、いらなくなったら塊で削除することができて、あとで瞬時にアクセス可能なようにきれいに並べかえられている構造

なんとなくわかってきたでしょう？

# 6. 多次元配列変数操作法応用

5 では、多次元配列変数操作を要素別に説明しましたが、この構造を使ってどんなことができるのかを例を示しながら、説明します。

## 6.1. 注文表

以下のような簡単な注文表を表示する処理を考えてみます。

注文表

注文番号 O1
お客様名 日本 太郎
ご住所 〒 100-0001 東京 丸の内

| 連番 | 商品名 | 単価 | 個数 | 合計 |
|---|---|---|---|---|
| 1 | PC001 | 300 | 1 | 300 |
| 2 | MNT001 | 500 | 3 | 1,500 |
| | | | 総合計 | 1,800 |

## 6.2. データモデル

一般的なリレーショナルモデルで考えると、上記の注文表を表示するためには、注文テーブル、注文明細テーブル、顧客情報テーブル、製品情報テーブルが必要になると想定されます。

多次元配列変数でこれらのテーブルを表現する方法は様々ですが、たとえば、以下の様な構造が考えられます。
なお、注文テーブルと注文明細テーブルは、親子関係と捕らえることができますので、同一変数の多次元構造として表現することができます。
もちろん、リレーショナルモデルと同じようなフラットな構造も取ることは可能ですが、今回示す多次元構造で表現する方法には後述するアクセス効率に関する大きなメリットがあります。

$LB は、$ListBuild 関数で生成されるレコードを表しています。
予約領域は、今回の説明では使いませんが、元々今回の説明のために流用したデータ[1]に既に定義されていたものです。無視してください。
これらのデータは、データベースとして保存するということを想定しますので、永続変数として実装します。

1) 顧客データ

```
^Shop.CustomerD(ID) =
    $LB(予約領域,名前, $LB(町名,都市名,郵便番号))
```

2) 注文データ

```
^Shop.POrderD(ID) =
    $LB(予約領域,注文番号,$LB(町名,都市名,郵便番号),購入日,処理中フラグ,出荷先（顧客）ID)
```

3) 注文明細

```
^Shop.POrderD(ID,"Items",seqno) =
    $LB(予約領域,製品 ID, 販売単価,個数)
```

[1] 別ガイド「IRIS オブジェクト操作ガイド」で解説するクラス定義を利用しています。

## 6.3. 処理要件

ここでは、注文明細にある特定の製品コード（PC001 など）が含まれる注文の注文表を表示するという処理を考えてみたいと思います。

単純には、全ての注文データを全件走査して、注文明細データに該当条件に合うものを探すという方法が考えられます。

しかし、この方法では、注文数が数百万件にもおよぶと全件走査するための時間が長くなってしまいます。
そして、データ件数が増えれば増えるほど処理時間は長くなってしまいます。

これを避けるには、どうしたらよいでしょうか？

## 6.4. インデックス用多次元データ変数

リレーショナルデータベースの場合、インデックスを構築して高速化するという手法を取ります。
多次元配列変数も同じアプローチが可能です。
ここでは、違う多次元配列変数をインデックスとして定義します。
素早くデータにアクセスするためには、該当する注文が特定できればよいので、その製品コードに対応する注文 ID の対応表を多次元配列変数で表現してみます。
それには、無限の表現方法がありますが、例として、以下の構造が考えられます。

```
^Shop.POrderI("CustomerIndex",pid, orderid)
```

pid は、製品コードを表しています。
orderid の部分に注文 ID が入ります。

同一製品コードが複数の注文に含まれる可能性があるので、orderid は複数の可能性があります。

インデックスとしては、これだけの情報で事足りるので、インデックス構造用の多次元配列変数ノードは、通常値を持ちません。

### 6.5. プログラム例

処理要件を満たすプログラム例を以下に示します。

```
OrderList() public {
Read "Enter Product ID >> ",pid,!!
Set n = "" Do {
        Set n = $Order(^Shop.POrderI("CustomerIndex",pid, n))
        If n = "" Quit
        Set precord = $Get(^Shop.POrderD(n))
        Set cid = $List(precord,6)
        Set cname = $List(^Shop.CustomerD(cid),2)
        Set ordernumber = $List(precord,2)
        Set address = $List(precord,3)
      Write ?30,"注文表",!
      Write "注文番号 ",ordernumber,!
      Write "お客様名 ",cname,!
      Write "ご住所 ","〒 ",$List(address,3)," ",$List(address,2)," ",$List(address,1),!
      Write !
      Write "連番    商品名                    単価  個数        合計",!
      Write "-------------------------------------------------------",!
      Set total = 0,i = 1
      Set n2 = "" Do {
            Set n2 = $Order(^Shop.POrderD(n,"Items",n2))
            If n2 = "" Quit
            Set itemrecord = $Get(^Shop.POrderD(n,"Items",n2))
            Set product = $Get(^Shop.ProductD($List(itemrecord,2)))
            Set productname = $List(product,2)
            Set unitprice = $List(itemrecord,3)
            Set amount = $List(itemrecord,4)
            Write
$J(i,3),?8,productname,?30,$J($FN(unitprice,","),8),?40,$J(amount,3),?50,$J($FN(unitprice*amount,","),8),!
              Set total = total + (unitprice*amount)
              Set i = i + 1
    } While(n2'="")
    Write "-------------------------------------------------------",!
    Write ?41,"総合計 ",$J($FN(total,","),10),!
  } While(n'="")
}
```

## 7. トランザクション管理

IRIS は、グローバルを使用した本格的なトランザクション処理を実装するのに必要な基本的な操作を提供します。
トランザクション用のコマンドには、TSTART、TCOMMIT、TROLLBACK があり、それぞれトランザクションの開始を定義する、トランザクションをコミットする、トランザクションを異常終了し、トランザクション開始してからグローバルに行われた変更を元に戻す操作を行います。

例えば、以下の ObjectScript コードは、トランザクションの開始を定義し、複数のグローバルノードを設定した後、ok という変数の値によって、そのトランザクションをコミットまたはロールバックします。

```
TSTART
Set ^Data(1) = "Apple"
Set ^Data(2) = "Berry"
If (ok) {
TCOMMIT
}
Else {
TROLLBACK
}
```

## 7.1. ロックとトランザクション

隔離されたトランザクション、つまりトランザクションがコミットする前に変更されたデータを他プロセスから見えないようにするには、ロックを使用する必要があります。
ObjectScript の中では、LOCK コマンドを使って直接ロックの取得や開放を行うことができます。
ロックは、取り決めとして、あるデータ構造（永続オブジェクトに使用されるものなど）に対して、ロックを必要とする全てのコードが同じ論理的なロック参照を使う必要があります。（例えば、LOCK コマンドが同じグローバルノード名を使用するなど）

トランザクションの中では、ロックは、特別な振る舞いを持ちます。

トランザクションの途中で獲得されたロックは、そのトランザクションの終了まで開放されません。

何故そうなのかを見るために、典型的なトランザクションによって実行されるアクションを見てみましょう。

1. **TSTART** を使用してトランザクションを開始
2. 変更したい 1 つまたは複数ノードに対してロックを獲得。 これは、通常"書き込み"ロックと呼ばれる。
3. 1 つまたは複数ノードを変更する。
4. ロックを開放する。トランザクション内なので、この時点では、これらのロックは実際には開放されない。
5. **TCOMMIT** を使用して、トランザクションをコミットする。この時点で以前のステップで開放された全てのロックが実際に開放される。

他プロセスが、このトランザクションに関わったノードを参照したいが、コミットされていない変更を見たくない時には、ノードからデータを読む前に単純に 1 つのロックを検査します（"読み取り"ロックと呼ぶ）。
書き込みロックは、トランザクションの最後まで保持されるので、読み取りプロセスは、トランザクションが完了（コミットまたはロールバック）するまでそのデータを参照しません。
大抵のデータベース管理システムもトランザクションの隔離性を提供するために同じようなメカニズムを使用します。
IRIS のユニークな点は、開発者がこのメカニズムを利用可能な点です。
トランザクションをサポートしながら、アプリケーションに特有なデータベース構造を作成することができます。

## 7.2. ネストトランザクション

ObjectScript は、特殊変数$TLEVEL を維持管理します。
これは、**TSTART** コマンドが何回呼ばれたかを追跡します。
*$TLEVEL* は、値 0 から始まり、**TSTART** の呼び出し毎に 1 を加えます。
一方*$TLEVEL* の値は、**TCOMMIT** の呼び出し毎に 1 を減じます。
**TCOMMIT** の呼び出しにより*$TLEVEL* が 0 に戻ると、トランザクションは終了します。
**TROLLBACK** コマンドの呼び出しは、いつも現トランザクションを終了し、*$TLEVEL* の値に関わらず *$TLEVEL* を 0 に戻します。

この振る舞いにより、アプリケーションの中で、それ自身がトランザクションを含むコードのまわりにトランザクションで囲むことができます。

## 7.3. 同時性の管理

1 つのグローバルノードの設定と取得の操作は、アトミックです。首尾一貫した結果を得ることがいつも保証されています。
複数ノードに対する操作またはトランザクションの隔離性を制御する操作のために ObjectScript は、ロックの取得と開放の機能を持っています。
ロックは、IRIS ロックマネージャが管理します。
ObjectScript の中では、 LOCK コマンドで直接ロックの獲得と開放ができます。

# 8. 多次元データモデルの設計に関する FAQ

## 8.1. 複数のデータの関連

グローバル変数のデータモデルでも、リレーショナルモデルと同様に、複数のデータの関連を表すことが可能です。

例えば、患者の診察データに対して、診察を行った医師を結びつけるとします。

その場合、次のように各患者の診察データに医師の ID を保持し、別のグローバル変数である、^Doctor に医師 ID をキーとする別の構造を作ります。

^Patient({患者 ID}, {診察日}) = {所見}^{医師 ID}
^Doctor({医師 ID})={診療課}^{氏名}

これを図で表すと次のようなイメージとなります。

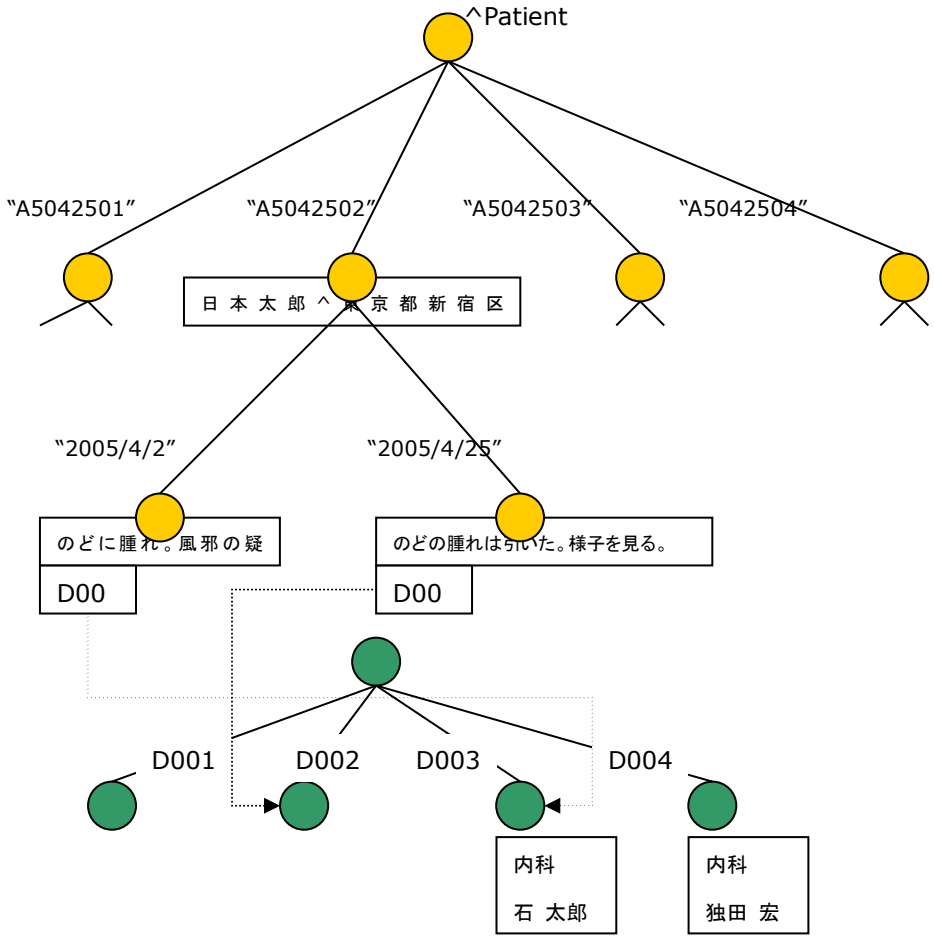


**図 3 複数のデータの関連（例：^Patient）**

## 8.2. インデックス

8.1 の例では、各患者のデータに診察のデータがありました。
この構造は、特定の患者からデータを検索するには適した構造ですが、例えば、特定日付に誰が診察を受けたかを調べるには効率が悪くなります。
それは次の図のように、日付のレベルのキーを全て「なめて」条件に合致する日付を捜さなければならないからです。

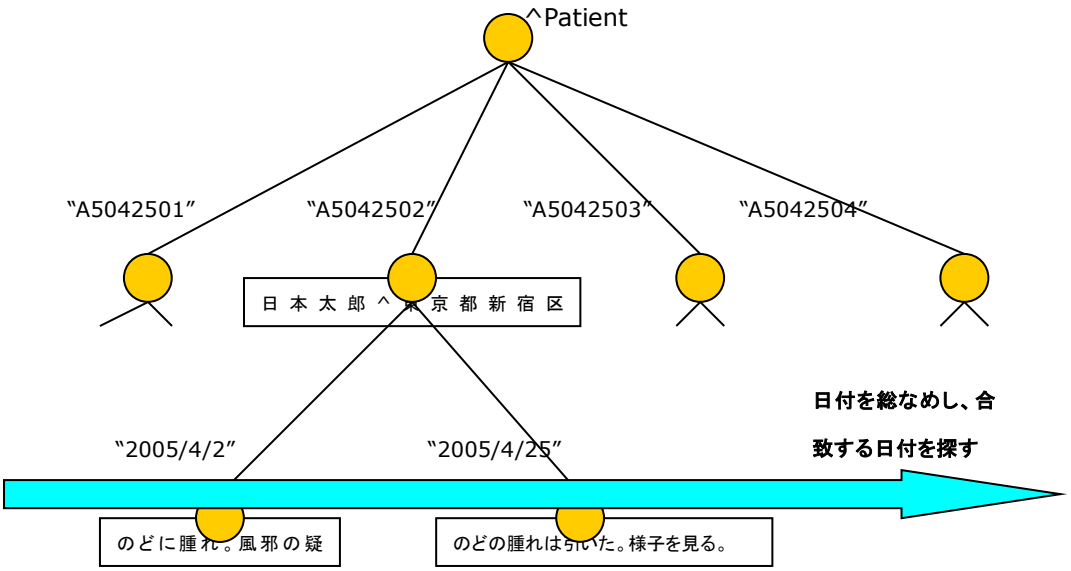


**図 4 ^Patient を利用して特定日付の診察を調査する場合のイメージ**

では、この検索の効率を上げるにはどうすればいいでしょうか？
多次元データモデルでは、このような場合、別のグローバル変数を定義し、それをインデックスとして利用するのがひとつの方法です。

インデックスとなるグローバル変数は次のような構造になります。

```
^PatientIdx("DateIdx", {診察日}, {患者 ID}) = ""
```

ここで一つ目のキーは日付のインデックスであることを示すために"DateIdx"という固定の文字列になっています。
2 番目のキーは診察日となっています。
診察日が変数項目の最初のキーとなっていることで、診察日による検索に備えています。

そして、患者 ID が 3 番目のキーとなっています。具体的なデータの例は次の通りです。

^PatientIdx(“DataIdx”, “2005/4/2”, “A5042502”) = ””
^PatientIdx(“DataIdx”, “2005/4/2”, “A5042601”) = ””
^PatientIdx(“DataIdx”, “2005/4/3”, “A4122205”) = ””
^PatientIdx(“DataIdx”, “2005/4/3”, “A5010101”) = ””

図で表すと次のようになります。

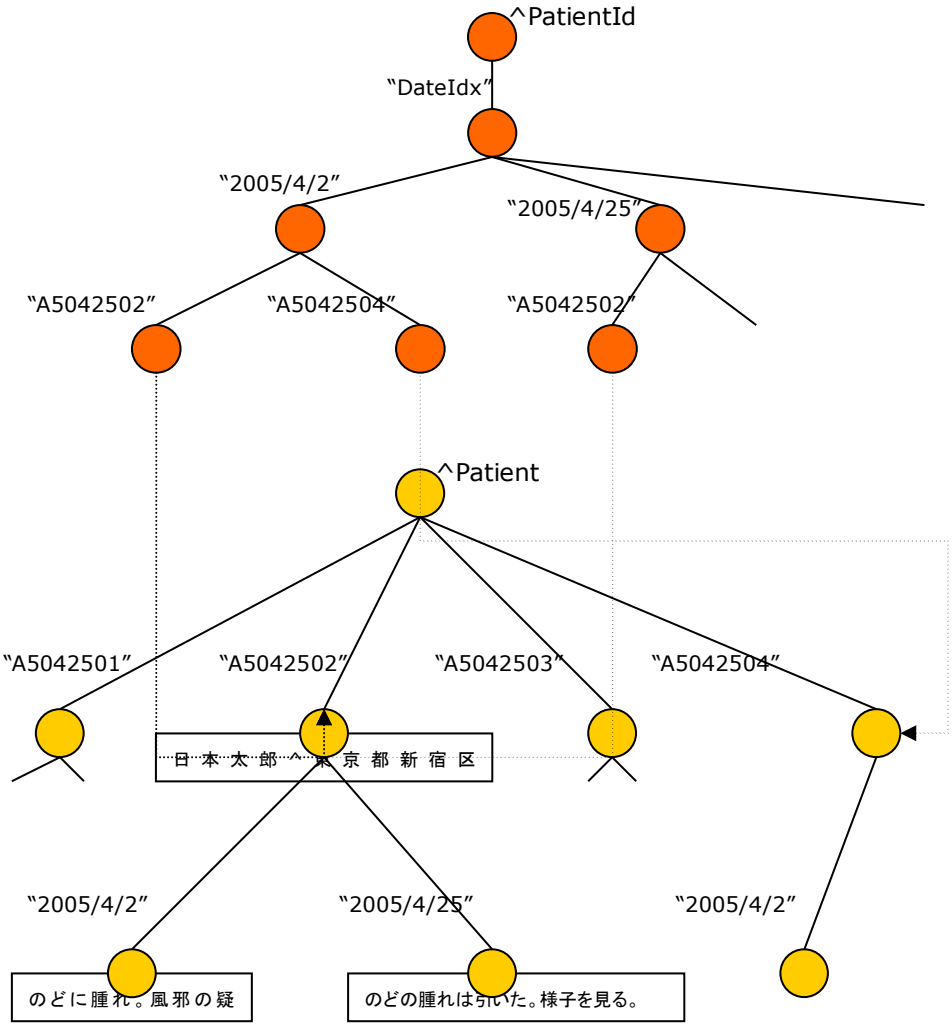


**図 5 インデックス：^PatientIdx**

# 9. 多次元配列変数内部構造

このような多次元配列の動的な構造は、内部的にはどうなっているのでしょうか？

構造に関して、メモリ上の変数と永続変数では、いろいろな違いがあります。

ここでは、データベースエンジンとしての多次元配列構造に着目し、永続変数の構造に
注力して説明します。

## 9.1. データベース

永続的な多次元配列変数（以下グローバル変数）は、データベースと呼ばれるコンテナファイル内で維持管理されます。

## 9.2. B+-Tree

データベースの中で、グローバル変数は、B+-Tree といわれる構造で管理されます。
B+-Tree 構造では、情報アクセスのために処理されるレベルが、全サブツリー間で等しくなるように保たれています。
利点は、求めるデータレコードを求めるためのブロックアクセス数が最小限ですむところです。
B+-Tree において、データレコードを含むデータブロックをポイントしている全てのキーが予めソートされています。
また、キー領域とデータ領域が統合されています。B+-Tree は、一番上のディレクトリブロック、1 つまたは複数のポインタブロック、データ部分を保存するデータブロックといういくつかのブロックタイプから構成されます。
グローバル変数を B+-Tree に保存する前に IRIS は、個々の添え字を連結して、1 つの文字列にします。
これは、アクセス速度が添え字のレベル数に依存しないことを意味します。

この B+-Tree 構造には、以下のようなメリットがあります。

1) 新しいエントリが挿入される時に、そのエントリを並べ替える。B+-Tree は、常に並び替えられていて、オーバフロー領域が必要ないので、再編成する必要がない。

2) 同じ先頭文字を持つ連続する添え字は、差分の部分だけが保存されるので、キー圧縮できる（常に並べ替えられているからこそ可能）

3) 同一レベルの次のブロックに対するポインタを保持するので、データをキー順に連続処理する際（$Order 関数など）に上位レベルのポインタブロックに戻る必要がない。

4) B+-Tree 構造の全ブロックは、プロセスがそれらをディスクに読み取ると即座にメモリ上のキャッシュに展開される。
その後の近傍のグローバル変数ノードに対するアクセスは、そのメモリ上で解決される可能性が高く、ディスク I/O を削減できる。

## 9.3. 実際の内部的な動き

グローバル変数を挿入、削除すると内部的にどのような変化が起こるか見てみましょう。
先ほどの演習と同じデータで考えてみます。
準備として先ほどのデータを消します。

```
USER>Kill ^PRODUCT,^A
```

```
USER>for i=1:1:100000 set ^A(i)=$lb(i,"abcdef",34.55)
```

この状態で、物理的なブロックの中身がどのようになっているか見てみましょう。

まずディレクトリブロックから見てみたいと思います。

該当グローバルの先頭ポインタブロックがどこにあるかは、%GD ユーティリティで確認できます。Show detail? という問いかけに対して、Yes と入力して下さい。

```
USER>do ^%SYS.GD

Which globals? * => A
Show global mappings? No => y
Show global attributes? No => y
Device:
Right margin: 80 =>


Global Directory Display of USER
                                    5:04 PM  Mar 23 2011

Name                          Location                              Keep  Jrn   Ptr
Grth

A                                   c:¥intersystems¥iris¥mgr¥user¥  いいえ
                                                                    はい 175    50
                    Collation: IRIS standard
                ResourceName: %DB_%DEFAULT:RW

1 global listed
```

上記のリストで^A グローバルの先頭ポインタブロック番号は、Ptr の下の値、175 となります。
この値および以降の例にあるブロック番号の値は、お使いのシステム毎に違う可能性が高いので、システムに合った値を入力するよう注意してください。

ポインタブロックやデータブロックの中身を参照するツールとして、REPAIR ユーティリティというものが用意されています。なお、この名前が示すとおり、このツールは、データベースの内部を参照するだけでなく、内容を書き換えることも可能で、データベースの内容が破壊されることもあり得ますので、ここで記述している以上の操作は、なるべく行わないよう注意して下さい。
このユーティリティは、%SYS ネームスペース上でしか動作しません。
実行に先だって、%SYS への移動が必要です。

REPAIR ユーティリティ起動後、データセットの物理的なロケーションを指定します。
通常、USER ネームスペースがマップするデータセットは、以下の例のようになります。

```
USER>zn "%SYS"

%SYS>do ^REPAIR

***Block Repair/Examine Program***


Directory: : c:¥intersystems¥iris¥mgr¥ => c:¥intersystems¥iris¥mgr¥user
Global Directory Block: 3

Map Blocks: 2
Entering Block Repair Menu
Block #:
```

Block #:のところで、先ほど%SYS.GD で調べたブロック番号 175 を入力すると、以下のようなデータが表示されます。

```
Block #: 175
Block # 175                 Type: 70 TOP/BOTTOM POINTER
Link Block: 0               Offset: 2652
Count of Nodes: 328         Collate: 5
First Node: ^A
Last Node:   ^A(99736)
--more--
```

--more--のところで Enter を押すと、さらに以下のようなデータが表示されます。

| # | Node | POINTER |
|---|---|---|
| 1 | ^A | 174 |
| 2 | ^A(306) | 2303 |
| 3 | ^A(611) | 2304 |

＜途中省略＞

| | | |
|---|---|---|
| 321 | ^A(97601) | 2838 |
| 322 | ^A(97906) | 2839 |
| 323 | ^A(98211) | 2840 |
| 324 | ^A(98516) | 2841 |
| 325 | ^A(98821) | 2842 |
| 326 | ^A(99126) | 2843 |
| 327 | ^A(99431) | 2844 |
| 328 | ^A(99736) | 2845 |

この表示の意味は、
^A から^A(306)より前のグローバルノードの次のレベルの情報は、ブロック番号 174 に定義されている。
同様に^A(306)から^A(611)より前のグローバルノードの次のレベルの情報は、ブロック番号 2303 に定義されている。ということを示しています。

次にブロック番号 174 を指定すると、以下のような表示があらわれます。

```
Block Repair Function (Current Block 175): 174
Block # 174                 Type: 8 DATA
Link Block: 2303            Offset: 7356
Count of Nodes: 306         Collate: 5              Big String Nodes: 0
Pointer Length:1            Next Pointer Length:5   Diff Byte:Hex 22
Pointer Reference:          ^A
Next Pointer Reference: ^A(306)
Next pointer stored? Yes
```

Type を見ると、データブロックとなっています。
Link Block は、次の同一レベル、つまり次のデータブロックのブロック番号が入っています。
Pointer Reference は、このブロックに入っている先頭のグローバルノード情報が入っています。

--more--の所、Enter を押すと、以下のような情報が表示されます。

```
#     Node                  Data
1     ^A
2     ^A(1)                 □□□□□abcdef□□þ□□
3     ^A(2)                 □□□□□abcdef□□þ□□
4     ^A(3)                 □□□□□abcdef□□þ□□
5     ^A(4)                 □□□□□abcdef□□þ□□
6     ^A(5)                 □□□□□abcdef□□þ□□

   <途中省略>

302   ^A(301)               □□-□□□abcdef□□þ□□
303   ^A(302)               □□.□□□abcdef□□þ□□
304   ^A(303)               □□/□□□abcdef□□þ□□
305   ^A(304)               □□0□□□abcdef□□þ□□
306   ^A(305)               □□1□□□abcdef□□þ□
```

Block Repair Function (Current Block 174):

このブロックは、データブロックなので、グローバルノードとデータが表示されます。
データは、$LIST 形式のデータなので、正常に表示されません。
同様に、ブロック番号 2303 を指定すると、以下のような表示になります。

```
Block Repair Function (Current Block 174): 2303
Block # 2303                Type: 8 DATA
Link Block: 2304            Offset: 7352
Count of Nodes: 305         Collate: 5              Big String Nodes: 0
Pointer Length:5            Next Pointer Length:5   Diff Byte:Hex 27
Pointer Reference:          ^A(306)
Next Pointer Reference:  ^A(611)
Next pointer stored? Yes


--more--
```

いかがですか？

なんとなく構造理解できましたか？

この例では、データ件数が比較的少ないため、グローバルディレクトリ->ポインタブロック
->データブロックの 3 つのブロックアクセスで目的のデータにたどりつけるようになっています。
どのデータにアクセスするにも現時点では、3 つのブロックの読み取りでできる、これこそが B+-Tree の特徴です。

もっとデータ量を多くするとどうなるでしょうか？

100 万件作成してみます。

```
USER>Kill ^PRODUCT,^A

USER>for i=1:1:1000000 set ^A(i)=$lb(i,"abcdef",34.55)
```

100 万件になると、さすがに少し時間がかかります。10～30 秒待ってください。
%SYS.GD で先頭ポインタブロックを調べます。（通常は、さきほどの値とかわらないはずです。）

```
USER>do ^%SYS.GD

Which globals? * => A
Show global mappings? No => y
Show global attributes? No => y
Device:
Right margin: 80 =>

                              Global Directory Display of USER
                                   5:29 PM  Mar 23 2011

Name                          Location                              Keep  Jrn   Ptr
Grth

A                             c:¥intersystems¥iris¥mgr¥user2¥ いいえ
                                                                    はい 175    50
                        Collation: IRIS standard
                     ResourceName: %DB_%DEFAULT:RW

1 global listed
```

起動中の REPAIR を一旦終了します。（REPAIR はデータ変更のたびに、再実行が必要です。）
REPAIR は、Block 番号入力のプロンプトに対して QUIT を入力すると終了します。

再度、REPAIR を起動し、ブロック番号 175 を入力します。

```
%SYS>do ^REPAIR

***Block Repair/Examine Program***


Directory: : c:¥intersystems¥iris¥mgr¥ => c:¥intersystems¥iris¥mgr¥user
Global Directory Block: 3

Map Blocks: 2
Entering Block Repair Menu
Block #: 175
Block # 175                 Type: 66 TOP POINTER
Link Block: 0               Offset: 64
Count of Nodes: 4           Collate: 5
First Node: ^A
Last Node:   ^A(746031)
```

Type が TOP POINTER になりました。（10 万件の時には、TOP/BOTTOM POINTER）

これは、次のレベルのブロックがデータブロックではなく、次のレベルのポインタブロックであることを示しています。

--more--の所、Enter を押すと、以下のような情報が表示されます。

```
#      Node                    POINTER
1      ^A                      3540
2      ^A(248881)                3539
3      ^A(497456)                4357
4      ^A(746031)                5173


Block Repair Function (Current Block 175):
```

ブロック番号 3540 を入力してみます。

```
Block Repair Function (Current Block 175): 3540
Block # 3540                Type: 6 BOTTOM POINTER
Link Block: 3539            Offset: 6556
Count of Nodes: 816         Collate: 5
First Node: ^A
Last Node:   ^A(248576)

--more--

#      Node                      POINTER
1      ^A                        174
2      ^A(306)                   2303

   <途中省略>

813   ^A(247661)                 3330
814   ^A(247966)                 3331
815   ^A(248271)                 3332
816   ^A(248576)                 3333


Block Repair Function (Current Block 3540):
```

ブロック番号 3330 を入力してみると、

```
Block Repair Function (Current Block 3540): 3330
Block # 3330                Type: 8 DATA
Link Block: 3331            Offset: 7352
Count of Nodes: 305         Collate: 5                Big String Nodes: 0
Pointer Length:6            Next Pointer Length:6     Diff Byte:Hex 5E
Pointer Reference:          ^A(247661)
Next Pointer Reference: ^A(247966)
Next pointer stored? Yes


--more--
```

```
#     Node                Data
1     ^A(247661)           □□mÇ□□□abcdef□□þ□□
2     ^A(247662)           □□nÇ□□□abcdef□□þ□□
3     ^A(247663)           □□oÇ□□□abcdef□□þ□□
4     ^A(247664)           □□pÇ□□□abcdef□□þ□□

   <途中省略>

302   ^A(247962)           □□□È□□□abcdef□□þ□□
303   ^A(247963)           □□□È□□□abcdef□□þ□□
304   ^A(247964)           □□□È□□□abcdef□□þ□□
305   ^A(247965)           □□□È□□□abcdef□□þ□□
```

Block Repair Function (Current Block 3330):

このように、データ件数が多くなったので、グローバルディレクトリ->先頭ポインタブロック->ボトムポインタブロック->データブロックの 4 つのブロックアクセスで目的のデータにたどりつけるようになりました。

さらにデータが増えると、ポインタブロックのレベルが 2 段から 3 段あるいは 4 段と増えていきます。
しかし、レベルの段の増加は、次第に緩慢になっていきます。データ件数に正比例する形で増えるわけではありません。
実際には、全体で 4 段以上になるケースは、かなり大規模データベース（テラバイト級）の場合となります。

以上は、非常に単純な構造のデータの場合でしたが、もう少し複雑な場合を見てみましょう。

以前と同様に以下の一連のコマンドをターミナルで実行します。

```
Kill ^PRODUCT
Set ^PRODUCT("A0001")="ポロシャツ"
Set ^PRODUCT("A0001","SS")=2000
Set ^PRODUCT("A0001","SS","赤")=10
Set ^PRODUCT("A0001","SS","茶")=10
Set ^PRODUCT("A0001","M")=3000
Set ^PRODUCT("A0001","M","赤")=10
Set ^PRODUCT("A0001","M","青")=5
Set ^PRODUCT("A0001","L")=3500
Set ^PRODUCT("A0001","L","赤")=7
Set ^PRODUCT("A0001","L","青")=20
Set ^PRODUCT("A0001","L","オレンジ")=3
Set ^PRODUCT("A0001","LL")=3800
Set ^PRODUCT("A0001","LL","赤")=5
Set ^PRODUCT("A0001","LL","青")=9
Set ^PRODUCT("A0001","LL","白")=3
```

%GD を実行し、^PRODUCT の先頭ポインタブロックを求めます。

```
USER>do ^%SYS.GD

Which globals? * => PRODUCT
Show global mappings? No => y
Show global attributes? No => y
Device:
Right margin: 80 =>

                        Global Directory Display of USER
                              5:53 AM  Mar 24 2011

Name                          Location                          Keep  Jrn   Ptr
Grth

PRODUCT                       c:¥intersystems¥iris¥mgr¥user¥    いいえ
                                                                はい 201    50
                    Collation: IRIS standard
                 ResourceName: %DB_USER:RW

1 global listed
```

REPAIR を実行し、ブロック番号 201 を入力します。

```
%SYS>do ^REPAIR

***Block Repair/Examine Program***


Directory: : c:¥intersystems¥iris¥mgr¥ => c:¥intersystems¥iris¥mgr¥user
Global Directory Block: 3

Map Blocks: 2
Entering Block Repair Menu
Block #: 201
Block # 201                 Type: 70 TOP/BOTTOM POINTER
Link Block: 0               Offset: 40
Count of Nodes: 1           Collate: 5

--more--

#      Node                       POINTER
1      ^PRODUCT                     200


Block Repair Function (Current Block 201):
```

ブロック 200 がデータブロックになります。

```
Block Repair Function (Current Block 201): 200
Block # 200                 Type: 8 DATA
Link Block: 0               Offset: 244
Count of Nodes: 16          Collate: 5              Big String Nodes: 0
Pointer Length:7            Next Pointer Length:0   Diff Byte:Hex 0
Pointer Reference:          ^PRODUCT
Next Pointer Reference:
Next pointer stored? No


--more--

#     Node                          Data
1     ^PRODUCT
2     ^PRODUCT("A0001")             ポロシャツ
3     ^PRODUCT("A0001","L")     3500 *
4     ^PRODUCT("A0001","L","オレンジ") 3 *
5     ^PRODUCT("A0001","L","赤") 7 *
6     ^PRODUCT("A0001","L","青") 20 *
7     ^PRODUCT("A0001","LL")    3800 *
8     ^PRODUCT("A0001","LL","白") 3 *
9     ^PRODUCT("A0001","LL","赤") 5 *
10    ^PRODUCT("A0001","LL","青") 9 *
11    ^PRODUCT("A0001","M")     3000 *
12    ^PRODUCT("A0001","M","赤") 10 *
13    ^PRODUCT("A0001","M","青") 5 *
14    ^PRODUCT("A0001","SS")    2000 *
15    ^PRODUCT("A0001","SS","茶") 10 *
16    ^PRODUCT("A0001","SS","赤") 10 *


Block Repair Function (Current Block 200):
```

設定されたデータが照合順に格納されていることがわかると思います。

なお、REPAIR ユーティリティの表示上（人が見てわかりやすい形式という意味で）、グローバルノードの部分は、ノード毎に添え字を全て表示していますが、実際には、さきほど少しふれたように、差分しかブロック内では格納しないようになっています。

例えば、

```
2       ^PRODUCT("A0001")          ポロシャツ
3       ^PRODUCT("A0001","L")      3500 *
```

の部分では、3 の所は、差分である"L"の情報しか持ちません。

REPAIR ユーティリティが情報を補って表示してくれるわけです。

今度はデータを追加してみて、ブロック構造がどうなるか見てみましょう。

データを追加してみます。

```
Set ^PRODUCT("A0001","M","水色")=4
Set ^PRODUCT("A0001","XL","水色")=4
```

REPAIR で確認してみます。
%SYS>do ^REPAIR

***Block Repair/Examine Program***

Directory: : c:¥intersystems¥iris¥mgr¥ => c:¥intersystems¥iris¥mgr¥user
Global Directory Block: 3

Map Blocks: 2
Entering Block Repair Menu
Block #: 201
Block # 201 Type: 70 TOP/BOTTOM POINTER
Link Block: 0 Offset: 40
Count of Nodes: 1 Collate: 5

--more--

| # | Node | POINTER |
|---|---|---|
| 1 | ^PRODUCT | 200 |

Block Repair Function (Current Block 201): 200
Block # 200 Type: 8 DATA
Link Block: 0 Offset: 276
Count of Nodes: 18 Collate: 5 Big String Nodes: 0
Pointer Length:7 Next Pointer Length:0 Diff Byte:Hex 0
Pointer Reference: ^PRODUCT
Next Pointer Reference:
Next pointer stored? No

--more--

```
#     Node                          Data
1     ^PRODUCT
2     ^PRODUCT("A0001")             ポロシャツ
3     ^PRODUCT("A0001","L")     3500 *
4     ^PRODUCT("A0001","L","オレンジ") 3 *
5     ^PRODUCT("A0001","L","赤") 7 *
6     ^PRODUCT("A0001","L","青") 20 *
7     ^PRODUCT("A0001","LL")    3800 *
8     ^PRODUCT("A0001","LL","白") 3 *
9     ^PRODUCT("A0001","LL","赤") 5 *
10    ^PRODUCT("A0001","LL","青") 9 *
11    ^PRODUCT("A0001","M")     3000 *
12    ^PRODUCT("A0001","M","水色") 4 *
13    ^PRODUCT("A0001","M","赤") 10 *
14    ^PRODUCT("A0001","M","青") 5 *
15    ^PRODUCT("A0001","SS")    2000 *
16    ^PRODUCT("A0001","SS","茶") 10 *
17    ^PRODUCT("A0001","SS","赤") 10 *
18    ^PRODUCT("A0001","XL","水色") 4 *
```

Block Repair Function (Current Block 200):

追加されたノードがきれいに並び替えられて、格納されていることが確認できます。
今度は、一部のデータを削除してみましょう。

Kill ^PRODUCT("A0001","LL")

REPAIR で確認してみます。
%SYS>do ^REPAIR

***Block Repair/Examine Program***

Directory: : c:¥intersystems¥iris¥mgr¥ => c:¥intersystems¥iris¥mgr¥user
Global Directory Block: 3

Map Blocks: 2
Entering Block Repair Menu
Block #: 201
Block # 201 Type: 70 TOP/BOTTOM POINTER
Link Block: 0 Offset: 40
Count of Nodes: 1 Collate: 5

--more--

# Node POINTER
1 ^PRODUCT 200

Block Repair Function (Current Block 201): 200
Block # 200 Type: 8 DATA
Link Block: 0 Offset: 224
Count of Nodes: 14 Collate: 5 Big String Nodes: 0
Pointer Length:7 Next Pointer Length:0 Diff Byte:Hex 0
Pointer Reference: ^PRODUCT
Next Pointer Reference:
Next pointer stored? No

--more--

```
#     Node                        Data
1     ^PRODUCT
2     ^PRODUCT("A0001")           ポロシャツ
3     ^PRODUCT("A0001","L")     3500 *
4     ^PRODUCT("A0001","L","オレンジ") 3 *
5     ^PRODUCT("A0001","L","赤") 7 *
6     ^PRODUCT("A0001","L","青") 20 *
7     ^PRODUCT("A0001","M")     3000 *
8     ^PRODUCT("A0001","M","水色") 4 *
9     ^PRODUCT("A0001","M","赤") 10 *
10    ^PRODUCT("A0001","M","青") 5 *
11    ^PRODUCT("A0001","SS")    2000 *
12    ^PRODUCT("A0001","SS","茶") 10 *
13    ^PRODUCT("A0001","SS","赤") 10 *
14    ^PRODUCT("A0001","XL","水色") 4 *
```

Block Repair Function (Current Block 200):

LL 配下のデータが全て消され、M のデータ以降が前に移動しているのがわかります。

挿入、削除が行われても、きれいに並べかえられているのです。
挿入の場合、

ノードが追加されることによって、該当データブロックに全て入りきらずに違うデータブロックに後ろの部分をずらす必要がある。

削除の場合には、

データブロックの中身が空になるので、それを前後のデータブロックと併合する。

など、実際はもっと複雑なケースがありますが、いずれにしてもどういう場合にも、ちゃんときれいに並び替えられる操作は確実に行われます。
データが断片化することがないのです。