記事
Toshihiko Minamoto · 2020年12月21日 9m read

$LIST のフォーマットと%DynamicArray 、%DynamicObject クラス

$LIST のフォーマットと%DynamicArray、%DynamicObject クラス

IRIS には、様々なデータ値を含むシーケンスを作成する方法がいくつかあります (以前は Cache にもありました)。 長年に渡り使用されているデータシーケンスの 1 つに $LIST の文字列があります。 より最近のデータシーケンスには %DynamicArray クラスと %DynamicObject クラスがあり、両者ともに JSON の文字列表現に対応する IRIS サポートの一部となっています。 これら 2 つのシーケンスにはそれぞれ非常に異なるトレードオフがあります。

$LIST の文字列形式

$LIST 形式は、かつてメモリアドレスのスペースが小さいだけでなく、ディスクドライブも小さく、読み取り速度が遅かった時代に考案されました。 $LIST の形式は、複数の異なるデータ型で構成されるシーケンスをバイト数を可能な限り抑えながら 8 ビットの一般的な文字列にパッキングするためにデザインされました。

$LIST のシーケンスは、ObjectScript の $LISTBUILD 関数を使って作成します。

$LIST の文字列の最も重要な機能は、データを 8 ビットの値で構成される最小のシーケンスにぎっしりパッキングできるという点です。 $LIST の文字列には、ObjectScript の様々なデータ型の複数の異なる表現を含めることができます。 それらのデータ型には、文字列型 (ObjectScript では引用符で囲んだ文字列リテラルを使って作成)、10 進浮動小数点型 (ObjectScript では数値リテラルを使って作成)、および IEEE 規格の 2 進浮動小数点型 (ObjectScript では $DOUBLE 関数を数値式に適用して作成) が含まれます。 ObjectScript の oref 型は $LIST の文字列によってサポートされていません。 $LIST の要素が作成されるとき、これらの値の内部表現に対しバイナリとバイトの非常にシンプルな圧縮化が行われます。

$LIST の文字列の 2 つ目の重要な機能は、これらの 8 ビットの値を IRIS インスタンスまたは転送メディアのエンディアン特性 (ビッグなのかリトルなのか) を気にせずに、両立させながら同インスタンス間を送信できるという点です。 特に明記すべきは、ビッグエンディアンを使うデータベースとリトルエンディアンを使うデータベースの間で $LIST の文字列を移動させる際に、$LIST のコンポーネントの値が変更されないという点です。

データのパッキングと転送機能に次いで重要なのがパフォーマンスです。 すべての $LIST オペレーション ($LISTVALID を除く) は、実行するマシンインストラクションの数を最小限に抑えようとします。 $LIST の構造が無効なために、セグメントフォールトや他のシステム例外が起り得る場合を除き、$LIST オペレーションは $LIST のデータ構造が有効であるという想定の基に実行されます。 $LIST オペレーションは、それを有効性を確認するためのインストラクションは実行しません。

空の $LIST は空の文字列です。 $LIST の文字列を連結するには、文字列の一般的な連結方法を用います。

$LIST の文字列をタイトにパッキングするということは、$LIST の **i** 個目の要素 (**$LIST(ListString,i)**) に直接ジャンプするのに効果的な情報はなく、先行する $LIST の要素をまず最初にスキャンする必要があることを意味します。 $LIST のすべての要素をスキャンする場合は、以下のコードを ***使ってはいけません***

```
Set N=$LISTLENGTH(ListString)
For i=1:1:N {
   Write $LIST(ListString,i),!
```

```
}
```

なぜなら、上の ObjectScript コードは、$LIST の値を O(N**2) の時間計算量でループするからです。
代わりに以下を使います。

```
Set P=0
While $LISTNEXT(ListString,P,value) {
    Write value,!
}
```

このコードは $LIST の値を O(N) の時間計算量でループします。

%DynamicArray クラス

最近では、%DynamicArray クラス (および %DynamicObject クラス) が作成されています。
今は、はるかに大きなメモリを使用できるようになり (現在のメモリは $LIST の文字列がデザインされた時代のディスクドライブよりも大きくなっている)、ディスクドライブも大幅にサイズアップしています。
今は構造化されたデータを異なるシステム間で送信するための標準的な形式があります。 これらには、XML や JSON が含まれます。 %DynamicArray クラス (および %DynamicObject クラス) には、JSON の値と ObjectScript の値 (oref 型の値も含む) を正確に表現する機能があります。 JSON の値と ObjectScript の値は非常によく似ている上に、1 つの種類の値が別の種類の値に変換されても、形式はほとんど変わりません (例外は、ObjectScript の oref 型の値で、JSON 特有の表現には変換できません)。

JSON の仕様では、JSON 配列リテラルは角括弧で囲まれると説明されています。 例えば、 **[0.1,"One tenth",2.99792E8,"speed of light in m/s"]** のように囲みます。 JSON のオブジェクトリテラルは、中括弧で囲まれます。 例えば、 **{"Name":"IRIS", "Size":64}** のように囲みます。
ObjectScript 言語では、JSON の配列リテラルとオブジェクトリテラルを使用できる他、JSON 配列または JSON オブジェクトが持つ要素の値を丸括弧で囲み、ObjectScript のランタイム式として使用できるという拡張機能が 1 つあります。 例えば、 **[(.1),("One " _"tenth"),(2.99793*1000)]** のようにできます。 丸括弧の中では、JSON の構文の代わりに ObjectScript の構文が使用されることに注意してください。

$LIST の文字列と %DynamicArray オブジェクトにはいくつか相違点があります。 (%DynamicObject オブジェクトのプロパティは %DynamicArray オブジェクトのプロパティと似ているので、以下のディスカッションで %DynamicObjectについて毎回言及するのは控えます。)

%DynamicArray の最初の要素は、インデックス 0 である一方で、$LIST の最初の要素はインデックス 1 となります。

%DynamicArray 要素は、ObjectScript の式で評価されるか、JSON 文字列に変換されるまでは、それが持つ元々の JSON の値または元々の ObjectScript の値を正確に表現でき、変換オペレーションにより小さな変化が生じることは一切ありません。

$LIST の文字列の最大サイズ (結果的にその要素のサイズ) は、現時点で 3641144 文字というObjectScript 文字列の最大長により制限されています。 %DynamicArray の最大サイズ (結果的にその要素のサイズ) はIRIS プロセスのメモリ空間のサイズにのみ制限されます。 これを踏まえ、多数の***大きな***%DynamicArray オブジェクトを同時に持つことは避けてください。これは、仮想アドレス空間を無制限で使用すると、過度のメモリーページングを引き起こす可能性があり、システムのパフォーマンスに影響するためです。

$LIST の文字列は、ObjectScript の文字列を格納できる場所であれば、どこにでも格納できます。
これは、文字列の長さが 3641144 文字をオーバーしないローカル変数やグローバル変数を含んでいます。
%DynamicArray オブジェクトをグローバル変数、ストリーム、またはファイルに移動する前には、%DynamicArray を他のデータ型 (通常は、JSON の表現を使用する文字列) に変換する必要があります。 JSON 文字列を持つ ObjectScript の文字列を返すには、引数なしの %ToJSON() メソッドを使用できます。 しかし、%DynamicArrays が大きいと、ObjectScript のグローバル変数や ObjectScript 式の中に収まりきらないほど長い JSON

の文字列が生成される場合があります。 この場合、%ToJSON(output) メソッドを実行すると、%DynamicArray が JSON 文字列に変換され、%ToJSON メソッドの引数によって指定される %Stream もしくはファイルに送付されます。

新しい %DynamicArray を効率よく作成するには、ObjectScript コンストラクタを使ったり、**%FromJSON(input)** メソッドを呼び出したりすると良いでしょう。 %DynamicArray のコンポーネントを調べる際に %Get メソッドを使用するのも効率の良い方法です。 特に、**$LIST(ListVar,i)** を評価する場合とは違い、**DynArray.%Get(i)** の評価にかかる時間が '**i**' の値や **DynArray.%Size()** の値に左右されない点に効率の良さが伺えます。 例えば、ObjectScript を使った次のループを実行すると、

```
Set N=DynArray.%Size()
For i=O:1:N-1 {
    Write DynArray.%Get(i),!
}
```

%DynamicArray に含まれるすべての要素が O(N) の時間計算量で出力され、$LIST(ListVar,i) メソッドを使いアクセスした要素が出力されるループを実行した場合に発生する O(N**2) の時間計算量は避けることができます。

次のようなループを書くこともできます。

```
Set iter = DynObject.%GetIterator()
While iter.%GetNext(.key , .value ) {
    Write "key = "_key_" , value = "_value,!
}
```

このループは、'**DynObject**' の未定義の要素をスキップする上に、%DynamicObject の要素と %DynamicArray の要素を出力するのに便利でもあります。

しかし、既に作成済みの %DynamicArray の内部要素を変更するのに '**Do DynArray.%Set(i,newvalue)**' を使用すると、DynArray に割り当てられたメモリの圧縮がある程度必要になる可能性があるほか、様々な内部インデックスの変更が必要になると思われるため、処理に時間がかかる可能性があります。
配列要素を大幅に変更する場合は、ObjectScript の多次元配列変数を使ってデータを表す方が無難と言えます。それは、ObjectScript の多次元配列は、配列要素の変更、挿入、削除に必要な時間を最小限に短縮できるようデザインされているためです。

IRIS 2019.1 では、**%Get(key,default,type)** メソッドと **%Set(key,value,type)** メソッドの機能が拡張されています。引数 '**default**' と '**type**' は省略可能です。 '**default**' 引数には、指定された '**key**' を持つ要素が未定義である場合に、**DynObject.%Get(key,default)** が返す値が入ります。 '**type**' パラメータとして使用できる値に、**"stream"** があります。これを使うことで、文字列値要素が ObjectScript の文字列に収まりきらないほど大きい場合に、%DynamicArray/%DynamicObject 要素の値を %Stream オブジェクトとして取得することができます。 '**type**' パラメータには、**"json"** という値も使用できます。これは、ObjectScript の値に使用される表現への変換を防止する JSON 仕様に従ってフォーマットされた文字列にアクセスします。
詳細は、クラスリファレンスのウェブページをご覧ください。 サポートされる 'type' の値は、IRIS の今後のリリースでさらに追加されると思われます。

#JSON #ObjectScript #ベストプラクティス #Caché #InterSystems IRIS

ソースURL:
https://jp.community.intersystems.com/post/list-%E3%81%AE%E3%83%95%E3%82%A9%E3%83%BC%E3%83%9E%E3%83%83%E3%83%88%E3%81%A8dynamicarray-%E3%80%81dynamicobject-%E3%82%AF%E3%83%A9%E3%82%B9