Article
Steven Hobbs · Oct 8, 2019 7m read

$LIST string format and %DynamicArray and %DynamicObject classes

$LIST string format and %DynamicArray and %DynamicObject classes

IRIS, and previously Cache, contain several different ways to create a sequence containing a mixture of data values. A data sequence that has been available for many years is the $LIST string. Another more recent data sequence is the %DynamicArray class, which along with the %DynamicObject class, is part of the IRIS support for JSON string representation. These two sequences involve very different tradeoffs.

$LIST String Format

$LIST format was invented some time ago when memory address spaces were small and when disk drives were also small and slow. $LIST format was designed to pack a sequence of several different data types into an ordinary 8-bit string using as few bytes as possible.

A $LIST sequence can be created using the ObjectScript $LISTBUILD function.

The most important feature of $LIST strings is packing the data tightly into a minimized sequence of 8-bit values. The $LIST strings can contain several different representations of the various ObjectScript types. Those ObjectScript types include the string type (which can be created in ObjectScript by a quoted string literal), the decimal floating-point type (which can be created in ObjectScript by numeric literal) and the IEEE binary floating-point type (which can be created in ObjectScript by applying the $DOUBLE function to a numeric expression.) The ObjectScriopt oref type is not supported by $LIST strings. The internal representations for these values undergo a very simple binary byte compaction when a $LIST element is created.

A second important feature of $LIST strings is the ability to compatibly transmit these 8-bit values between IRIS instances without worrying about big-endian versus little-endian characteristics of the instances or of the transport media. In particular, a $LIST string can be moved between a big-endian data base and a little-endian data base without changing the values of the $LIST components.

After packing and transportability in importance comes performance. All of the $LIST operations (except $LISTVALID) try to execute the minimum number of machine instructions. Except in the cases when an invalid $LIST structure could lead to a segment fault or other system exception, the $LIST operations assume that the $LIST data structures are valid. The $LIST operations avoid executing extra instructions to test for validity.

The empty $LIST is an empty string. Two $LIST strings can be concatenated by using ordinary string concatenation.

The tight packing of $LIST strings means there are no additional data words to make it efficient to jump directly to the **i**-th $LIST element, **$LIST(ListString,i)**, without first scanning through the preceding $LIST elements. If you want to scan through all the elements of a $LIST you should *__NOT__* use:

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

because that ObjectScript code will loop over the $LIST values in O(N**2) time complexity. Instead use:

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

which will loop over the $LIST values in O(N) time complexity.

The %DynamicArray Class

More recently the %DynamicArray class (and the %DynamicObject class) have been created. We now have much larger memories (i.e., memories are now bigger than the disk drives that existed back when the $LIST string was designed) and we also have much larger disk drives. There are now standardized formats for sending structured data between different systems. These standards include XML and JSON. The %DynamicArray class (and %DynamicObject class) includes the capability of exactly representing JSON values as well as representing ObjectScript values (including the oref value.) JSON values and ObjectScript values are very similar and there is very little change in semantics when one kind of value is converted to the other kind of value (with the exception of the ObjectScript oref value which cannot be converted into JSON representation.)

The JSON standard describes that a JSON Array literal is enclosed in square brackets. E.g. **[0.1,"One tenth",2.99792E8,"speed of light in m/s"]**. A JSON Object literal is enclosed in curly brackets. E.g. **{"Name":"IRIS", "Size":64}**. The ObjectScript language accepts JSON Array and Object literals, with the addition of one extension--a value of an element of a JSON Array or JSON Object can be an ObjectScript run-time expression enclosed in round brackets. E.g. **[(.1),("One " _"tenth"),(2.99793*1000)]** . Note that inside the round parentheses ObjectScript syntax is used instead of JSON syntax.

There are differences between the $LIST string and the %DynamicArray object. (%DynamicObject objects have similar properties to %DynamicArray objects so we will not always mention %DynamicObjects in the following discussion.)

The first element of a %DynamicArray has index 0 while the first element of a $LIST has index 1.

Until a %DynamicArray element is evaluated in an ObjectScript expression or is converted into a JSON string, that element can exactly represent either its original JSON value or its original ObjectScript value without any minor changes due to the conversion operations.

The maximum size of a $LIST string (and consequently the sizes of its elements) is limited by the maximum length of an ObjectScript string, which is currently 3641144 characters. The maximum size of a %DynamicArray (and consequently the sizes of its elements) is limited only by the amount of memory space available to the IRIS process. As a result, avoid having a large number of *__huge__* %DynamicArray objects active at the same time as unlimited use of virtual address space can cause excessive memory paging which will hurt system performance.

A $LIST string can be stored anywhere an ObjectScript string can be stored. This includes local and global variables providing the string length does not exceed 3641144 characters. Before a %DynamicArray object can be moved into a global variable, into a stream or into a file, the %DynamicArray must be converted into another data type, usually a character string using JSON representation. The %ToJSON() method without an argument can be used to return an ObjectScript string containing the JSON string. However, large %DynamicArrays can generate JSON strings that are too long to fit in ObjectScript globals or ObjectScript expressions. In this case, evaluating the %ToJSON(output) method will convert the %DynamicArray to a JSON string that will be sent either to a %Stream or to a file which is specified by the argument to %ToJSON.

Creating a new %DynamicArray using ObjectScript constructors or calling the **%FromJSON(input)** method can be reasonably efficient. Examining the components of a %DynamicArray using the %Get method is also efficient. In particular, the execution time required to evaluate **DynArray.%Get(i)**, unlike evaluation of **$LIST(ListVar,i)**, is independent of the value of '**i**' and independent of the value of **DynArray.%Size()**. For example, the following ObjectScript loop:

```
Set N=DynArray.%Size()
For i=O:1:N-1 {
```

```
    Write DynArray.%Get(i),!
}
```

will print out all the elements in a %DynamicArray with complexity O(N) and without the O(N**2) complexity that would occur in the corresponding loop printing out the elements accessed by $LIST(ListVar,i).

You can also write the following loop:

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

which will skip the undefined elements of '**DynObject**' and which is also convenient for printing %DynamicObject elements as well as %DynamicArray elements.

However, using '**Do DynArray.%Set(i,newvalue)**' to modify an internal element of an already constructed %DynamicArray can be time-consuming because the memory allocated to DynArray may require some compaction and the various internal indexes will probably need some modification. If you are doing extensive modifications of array elements then you may be better off using an ObjectScript MultiDimensional array variable to represent your data since ObjectScript MultiDimensional arrays are designed to minimize the time needed to do array element modifications, insertions and deletions.

In IRIS 2019.1 there have been extensions made to the **%Get(key,default,type)** and **%Set(key,value,type)** methods. The arguments '**default**' and '**type**' are optional. The '**default**' argument contains the value returned by **DynObject.%Get(key,default)** when the element with the specified '**key**' is undefined. A possible value for the '**type**' parameter includes **"stream"** which allows you to get the value of a %DynamicArray/%DynamicObject element as a %Stream object in cases where a string valued element is too large to fit in an ObjectScript string. Another possible value for the '**type**' parameter is **"json"** which accesses a string formatted according to the JSON standard which prevents the conversions to the representations used by ObjectScript values. See the Class Reference web pages for more details. There should be additional 'type' values that will be supported in future IRIS releases.

#Best Practices #JSON #ObjectScript #Caché #InterSystems IRIS

Source URL:https://community.intersystems.com/post/list-string-format-and-dynamicarray-and-dynamicobject-classes