Article
Eduard Lebedyuk · Sep 10, 2018 4m read

Dynamic objects and JSON support in InterSystems products

Generally speaking, InterSystems products supported dynamic objects and JSON for a long while, but version 2016.2 came with a completely new implementation of these features, and the corresponding code was moved from the ObjectScript level to the kernel/C level, which made for a substantial performance boost in these areas. This article is about innovations in the new version and the migration process (including the ways of preserving backward compatibility).

Working with JSON

Let me start with an example. The following syntax works now and it ' s the biggest innovation in ObjectScript syntax:

```
Set object = { "property": "val", "property2": 2, "property3": null }
Set array = [ 1, 2, "string", true ]
```

As you can see, JSON is now an integral part of ObjectScript. So what happens when values are assigned this way? The " object " becomes an instance of the %Library.DynamicObject object, while " array " is an instance of %Library.DynamicArray. They are both dynamic objects.

Dynamic objects

Cache had dynamic objects for a while in the form of the %ZEN.proxyObject class, but now the code has been moved to the kernel for greater performance. All dynamic object classes are inherited from %Library.DynamicAbstractObject, which offers the following functionality:

- Getting an object from a JSON string, stream, file
- Output of an object in the JSON format to a string or variable, automatic detection of the output format depending on context
- Writing an object to a file in the JSON format
- Writing an object to a global
- Reading an object from a global

Transition from %ZEN.proxyObject

So you want to migrate from %ZEN.proxyObject/%Collection.AbstractIterator towards %Library.DynamicAbstractObject? This is a relatively simple task that can be completed in several ways:

- If you are not interested in compatibility with versions of Caché earlier than 2016.2, just use Ctrl+H thoughtfully and you ' ll be fine. Keep in mind, though, that indexes in arrays start with a zero now and you need to add $ to the names of system methods
- Use macros that transform the code into the necessary form during compilation depending on the version of Caché.
- Use an abstract class as a wrapper for corresponding methods

The use of the first method is obvious, so let ' s focus on the remaining two.

Macros

An approximate set of macros that work with either new or old dynamic objects class depending on the availability of %Library.AbstractObject.

Macros

property

```
#if $$$comClassDefined("%Library.dynamicAbstractObject")
    #define NewDynObj {}
    #define NewDynDTList []
    #define NewDynObjList $$$NewDynDTList
    #define Insert(%obj,%element) do %obj.%Push(%element)
    #define DynObjToJSON(%obj) w %obj.%ToJSON()
    #define ListToJSON(%obj) $$$DynObjToJSON(%obj)
    #define ListSize(%obj) %obj.%Size()
    #define ListGet(%obj,%i) %obj.%Get(%i-1)
#else
    #define NewDynObj ##class(%ZEN.proxyObject).%New()
    #define NewDynDTList ##class(%ListOfDataTypes).%New()
    #define NewDynObjList ##class(%ListOfObjects).%New()
    #define Insert(%obj,%element) do %obj.Insert(%element)
    #define DynObjToJSON(%obj) do %obj.%ToJSON()
    #define ListToJSON(%obj) do ##class(%ZEN.Auxiliary.jsonProvider).%ObjectToJSON(%o
bj)
    #define ListSize(%obj) %obj.Count()
    #define ListGet(%obj,%i) %obj.GetAt(%i)
#endif
#define IsNewJSON ##Expression($$$comClassDefined("%Library.DynamicAbstractObject"))
```

This code:

```
 Set obj = $$$NewDynObj
 Set obj.prop = "val"
 $$$DynObjToJSON(obj)

 Set dtList = $$$NewDynDTList
 Set a = 1
 $$$Insert(dtList,a)
 $$$Insert(dtList,"a")
 $$$ListToJSON(dtList)
```

Will compile into the following code for Cache version 2016.2+:

```
 set obj = {}
 set obj.prop = "val"
 w obj.%ToJSON()
 set dtList = []
 set a = 1
 do dtList.%Push(a)
 do dtList.%Push("a")
 w dtList.%ToJSON()
```

For previous versions, it will look like this:

```
set obj = ##class(%ZEN.proxyObject).%New()
set obj.prop = "val"
do obj.%ToJSON()
set dtList = ##class(%Library.ListOfDataTypes).%New()
set a = 1
do dtList.Insert(a)
do dtList.Insert("a")
do ##class(%ZEN.Auxiliary.jsonProvider).%ObjectToJSON(dtList)
```

Abstract class

The alternative is to create a class that abstracts the dynamic object being used. For example:

```
Class Utils.DynamicObject Extends %RegisteredObject
{
/// A property storing a true dynamic object
Property obj;
Method %OnNew() As %Status
{
    #if $$$comClassDefined("%Library.DynamicAbstractObject")
        Set ..obj  = {}
    #else
        Set ..obj  = ##class(%ZEN.proxyObject).%New()
    #endif
    Quit $$$OK
}
/// Getting dynamic properties
Method %DispatchGetProperty(pProperty As %String) [ Final ]
{
    Quit ..obj.%DispatchGetProperty(pProperty)
}
/// Setting dynamic properties
Method %DispatchSetProperty(pProperty As %String, pValue As %String) [ Final ]
{
    Do ..obj.%DispatchSetProperty(pProperty,pValue)
}
/// Converting to JSON
Method ToJSON() [ Final ]
{
    Do ..obj.%ToJSON()
}
}
```

Using this class is completely identical to working with a regular one:

```
Set obj = ##class(Utils.DynamicObject).%New()
Set obj.prop = "val"
Do obj.ToJSON()
```

What to choose

It ' s totally your call. The option with a class looks more conventional, the one with macros is a bit faster thanks to the absence of intermediate calls. For my MDX2JSON project, I chose the latter option with macros. The transition was fast and smooth.

JSON performance

The speed of JSON generation went up dramatically. The MDX2JSON project has JSON generation tests that you can download to see the performance boost with your own eyes!

Conclusion

New dynamic objects and improvements in JSON support will help your applications work faster.

Links

- Documentation

#Beginner #Caché #JSON #Tutorial