Article
Eduard Lebedyuk · Mar 24, 2017 9m read

Logging using macros in InterSystems IRIS

In my previous article, we reviewed possible use-cases for macros, so let ' s now proceed to a more comprehensive example of macros usability. In this article we will design and build a logging system.

Logging system

Logging system is a useful tool for monitoring the work of an application that saves a lot of time during debugging and monitoring. Our system would consist of two parts:

- Storage class (for log records)
- Set of macros that automatically add a new record to the log

Storage class

Let ' s create a table of what we need to store and specify when this data can be obtained – during compilation or at runtime. This will be required when working on the second part of the system - macros, where we will aim to have as many loggable details during compilation as possible:

Information	Obtained during
Event type	Compilation
Class name	Compilation
Method name	Compilation
Arguments passed to a method	Compilation
Line number in the cls source code	Runtime
Line number in the generated int code	Runtime
Username	Runtime
Date/Time	Runtime
Message	Runtime
IP address	Runtime

Let ' s create an App.Log class containing the properties from the table above. When an App.Log object is created, User Name, Date/Time and IP address properties are filled out automatically.

App.Log class:

```
Class App.Log Extends %Persistent
{

/// Type of event
Property EventType As %String(MAXLEN = 10, VALUELIST = ",NONE,FATAL,ERROR,WARN,INFO,S
```

```
TAT,DEBUG,RAW") [ InitialExpression = "INFO" ];

/// Name of class, where event happened
Property ClassName As %Dictionary.Classname(MAXLEN = 256);

/// Name of method, where event happened
Property MethodName As %String(MAXLEN = 128);

/// Line of int code
Property Source As %String(MAXLEN = 2000);

/// Line of cls code
Property SourceCLS As %String(MAXLEN = 2000);

/// Cache user
Property UserName As %String(MAXLEN = 128) [ InitialExpression = {$username} ];

/// Arguments' values passed to method
Property Arguments As %String(MAXLEN = 32000, TRUNCATE = 1);

/// Date and time
Property TimeStamp As %TimeStamp [ InitialExpression = {$zdt($h, 3, 1)} ];

/// User message
Property Message As %String(MAXLEN = 32000, TRUNCATE = 1);

/// User IP address
Property ClientIPAddress As %String(MAXLEN = 32) [ InitialExpression = {..GetClientAd
dress()} ];

/// Determine user IP address
ClassMethod GetClientAddress()
{
    // %CSP.Session source is preferable
    #dim %request As %CSP.Request
    If ($d(%request)) {
        Return %request.CgiEnvs("REMOTE_ADDR")
    }
    Return $system.Process.ClientIPAddress()
}
}
```

Logging macros

Usually, macros are stored in separate *.inc files containing their definitions. The necessary files can be included into classes using the Include MacroFileName command, which in this case will look as follows: Include App.LogMacro.

To start, let ’ s define the main macro that the user will add to their application ’ s code:

```
#define LogEvent(%type, %message) Do ##class(App.Log).AddRecord($$$CurrentClass, $$$C
urrentMethod, $$$StackPlace, %type, $$$MethodArguments, %message)
```

This macro accepts two input arguments: Event Type and Message. The Message argument is defined by the user,

but the Event Type parameter will require additional macros with different names that will automatically identify the event type:

```
#define LogNone(%message)          $$$LogEvent("NONE", %message)
#define LogError(%message)         $$$LogEvent("ERROR", %message)
#define LogFatal(%message)         $$$LogEvent("FATAL", %message)
#define LogWarn(%message)          $$$LogEvent("WARN", %message)
#define LogInfo(%message)          $$$LogEvent("INFO", %message)
#define LogStat(%message)          $$$LogEvent("STAT", %message)
#define LogDebug(%message)         $$$LogEvent("DEBUG", %message)
#define LogRaw(%message)           $$$LogEvent("RAW", %message)
```

Therefore, in order to perform logging, the user only needs to place the $$$LogError("Additional message") macro in the application code.
All we need to do now is to define the $$$CurrentClass, $$$CurrentMethod, $$$StackPlace, $$$MethodArguments macros. Let ' s start with the first three:

```
#define CurrentClass      ##Expression($$$quote(%classname))
#define CurrentMethod     ##Expression($$$quote(%methodname))
#define StackPlace        $st($st(-1),"PLACE")
```

%classname, %methodname variables are described in the documentation. The $stack function returns INT code line number. To convert it into CLS line number we can use this code.

Let's use the %Dictionary package to get a list of method arguments and their values. It contains all the information about the classes, including method descriptions. We are particularly interested in the %Dictionary.CompiledMethod class and its FormalSpecParsed property, which is a list:

```
$lb($lb("Name","Classs","Type(Output/ByRef)","Default value "),...)
```

corresponding to the method signature. For example:

```
ClassMethod Test(a As %Integer = 1, ByRef b = 2, Output c)
```

will have the following FormalSpecParsed value:

```
$lb(
$lb("a","%Library.Integer","","1"),
$lb("b","%Library.String","&","2"),
$lb("c","%Library.String","*",""))
```

We need to make $$$MethodArguments macro expand into the following code (for the Test method):

```
"a="_$g(a,"Null")_"; b="_$g(b,"Null")_"; c="_$g(c,"Null")_";"
```

To achieve this, we have to do the following during compilation:

1. Get a class name and a method name
2. Open a corresponding instance of the %Dictionary.CompiledMethod class and get its FormalSpec property
3. Convert it into a source code line

Let's add corresponding methods to the App.Log class:

```
ClassMethod GetMethodArguments(ClassName As %String, MethodName As %String) As %Strin
g
{
    Set list = ..GetMethodArgumentsList(ClassName,MethodName)
    Set string = ..ArgumentsListToString(list)
    Return string
}

ClassMethod GetMethodArgumentsList(ClassName As %String, MethodName As %String) As %L
ist
{
    Set result = ""
    Set def = ##class(%Dictionary.CompiledMethod).%OpenId(ClassName _ "||" _ MethodNa
me)
    If ($IsObject(def)) {
        Set result = def.FormalSpecParsed
    }
    Return result
}

ClassMethod ArgumentsListToString(List As %List) As %String
{
    Set result = ""
    For i=1:1:$ll(List) {
        Set result = result _ $$$quote($s(i>1=0:"",1:"; ") _ $lg($lg(List,i))_"=")
        _ "_$g(" _ $lg($lg(List,i)) _ ","_$$$quote(..#Null)_")_"
        _$s(i=$ll(List)=0:"",1:$$$quote(";"))
    }
    Return result
}
```

Let ’ s now define the $$$MethodArguments macro as:

```
#define MethodArguments ##Expression(##class(App.Log).GetMethodArguments(%classname,%
methodname))
```

Use case

Next, let's create an App.Use class with a Test method to demonstrate the capabilities of the logging system:

```
Include App.LogMacro
Class App.Use [ CompileAfter = App.Log ]
{
/// Do ##class(App.Use).Test()
ClassMethod Test(a As %Integer = 1, ByRef b = 2)
{
    $$$LogWarn("Text")
}
}
```

As a result, the $$$LogWarn("Text") macro in the int code converts into the following line:

```
Do ##class(App.Log).AddRecord("App.Use","Test",$st($st(-1),"PLACE"),"WARN","a="_$g(a,
"Null")_"; b="_$g(b,"Null")_";", "Text")
```

Execution of this code will create a new App.Log record:

ID	Arguments	ClassName	ClientIPAddress	EventType	Message	MethodName	Source	SourceCLS	TimeStamp	UserName
1	a=1; b=2;	App.Use2	127.0.0.1	WARN	Text	Test	zTest+1^App.Use2.1 +1	App.Use:Test+1	2017-03-24 16:36:34	UnknownUser

Improvements

Having created a logging system, here's some improvement ideas:

- First of all, a possibility to process object-type arguments since our current implementation only logs object oref.
- Second, a call to restore the context of a method from stored argument values.

Processing of object-type arguments

The line that puts an argument value to the log is generated in the ArgumentsListToString method and looks like this:

```
"_$g(" _ $lg($lg(List,i)) _ ","_$$$quote(..#Null)_")_"
```

Let's do some refactoring and move it into a separate GetArgumentValue method that will accept a variable name and class (all of which we know from FormalSpecParsed) and output a code that will convert the variable into a line. We'll use existing code for data types, and objects will be converted into JSON with the help of SerializeObject (for calling from the user code) and WriteJSONFromObject (for converting an object into JSON) methods:

```
ClassMethod GetArgumentValue(Name As %String, ClassName As %Dictionary.CacheClassname
) As %String
{
    If $ClassMethod(ClassName, "%Extends", "%RegisteredObject") {
        // it's an object
        Return "_##class(App.Log).SerializeObject("_Name _ ")_"
    } Else {
        // it's a datatype
        Return "_$g(" _ Name _ ","_$$$quote(..#Null)_")_"
    }
}

ClassMethod SerializeObject(Object) As %String
{
    Return:'$IsObject(Object) Object
    Return ..WriteJSONFromObject(Object)
}

ClassMethod WriteJSONFromObject(Object) As %String [ ProcedureBlock = 0 ]
{
    Set OldIORedirected = ##class(%Device).ReDirectIO()
    Set OldMnemonic = ##class(%Device).GetMnemonicRoutine()
    Set OldIO = $io
    Try {
        Set Str=""
```

```
        //Redirect IO to the current routine - makes use of the labels defined below
        Use $io::("^"_$ZNAME)

        //Enable redirection
        Do ##class(%Device).ReDirectIO(1)

        Do ##class(%ZEN.Auxiliary.jsonProvider).%ObjectToJSON(Object)
    } Catch Ex {
        Set Str = ""
    }

    //Return to original redirection/mnemonic routine settings
    If (OldMnemonic '= "") {
        Use OldIO::("^"_OldMnemonic)
    } Else {
        Use OldIO
    }
    Do ##class(%Device).ReDirectIO(OldIORedirected)

    Quit Str

    // Labels that allow for IO redirection
    // Read Character - we don't care about reading
rchr(c)      Quit
    // Read a string - we don't care about reading
rstr(sz,to)  Quit
    // Write a character - call the output label
wchr(s)      Do output($char(s))  Quit
    // Write a form feed - call the output label
wff()        Do output($char(12))  Quit
    // Write a newline - call the output label
wnl()        Do output($char(13,10))  Quit
    // Write a string - call the output label
wstr(s)      Do output(s)  Quit
    // Write a tab - call the output label
wtab(s)      Do output($char(9))  Quit
    // Output label - this is where you would handle what you actually want to do.
    // in our case, we want to write to Str
output(s)    Set Str = Str_s Quit
}
```

A log entry with an object-type argument looks like this:

2	a=1; b={ "prop1":123, "prop2":"abc" };	App.Use	127.0.0.1	WARN	User message	TestWithObjects	zTestWithObjects+1^App.Use.1 +1	App.Use:TestWithObjects+1	2017-03-24 16:40:56	UnknownUser

Restoring the context

The idea of this method is to make all arguments available in the current context (mostly in the terminal, for debugging). To this end, we can use the ProcedureBlock method parameter. When set to 0, all variables declared within such a method will remain available upon quitting the method. Our method will open an object of the App.Log class and deserialize the Arguments property.

```
ClassMethod LoadContext(Id) As %Status [ ProcedureBlock = 0 ]
{
    Return:'..%ExistsId(Id) $$$OK
    Set Obj = ..%OpenId(Id)
    Set Arguments = Obj.Arguments
```

```
    Set List = ..GetMethodArgumentsList(Obj.ClassName,Obj.MethodName)
    For i=1:1:$Length(Arguments,";")-1 {
        Set Argument = $Piece(Arguments,";",i)
        Set @$lg($lg(List,i)) = ..DeserializeObject($Piece(Argument,"=",2),$lg($lg(Li
st,i),2))
    }
    Kill Obj,Arguments,Argument,i,Id,List
}

ClassMethod DeserializeObject(String, ClassName) As %String
{
    If $ClassMethod(ClassName, "%Extends", "%RegisteredObject") {
        // it's an object
        Set st = ##class(%ZEN.Auxiliary.jsonProvider).%ConvertJSONToObject(String,,.o
bj)
        Return:$$$ISOK(st) obj
    }
    Return String
}
```

This is how it looks in the terminal:

```
>zw
>do ##class(App.Log).LoadContext(2)
>zw

a=1
b=<OBJECT REFERENCE>[2@%ZEN.proxyObject]

>zw b
b=<OBJECT REFERENCE>[2@%ZEN.proxyObject]
+----------------- general information ---------------
|      oref value: 2
|      class name: %ZEN.proxyObject
| reference count: 2
+----------------- attribute values ------------------
|           %changed = 1
|     %data("prop1") = 123
|     %data("prop2") = "abc"
|             %index = ""
```

What ' s next?

The key potential improvement is to add another argument to the log class with an arbitrary list of variables created inside the method.

Conclusions

Macros can be quite useful for application development.

Questions

Is there a way to obtain line number during compilation?

Links

- Part I. Macros
- GitHub repository

#Best Practices #Compiler #Object Data Model #Caché #InterSystems IRIS