Artículo
Kurro Lopez · 16 jun, 2020 Lectura de 9 min

Logging usando macros en InterSystems Caché

En mi anterior artículo, revisamos los posibles casos de uso para macros, así que pasemos ahora a un ejemplo más completo de usabilidad de macros. En este artículo diseñaremos y crearemos un sistema de registro.

Sistema de registro

El sistema de registro es una herramienta útil para monitorear el trabajo de una aplicación que ahorra mucho tiempo durante la depuración y el monitoreo. Nuestro sistema constaría de dos partes:

- Clase de almacenamiento (para registros de anotaciones)
- Conjunto de macros que agregan automáticamente un nuevo registro al registro

Clase de almacenamiento

Vamos a crear una tabla de lo que necesitamos almacenar y especificar cuándo se pueden obtener estos datos, durante la compilación o en tiempo de ejecución. Esto será necesario cuando trabaje en la segunda parte del sistema: macros, donde buscaremos tener tantos detalles registrables como sea posible durante la compilación:

Información	**Obtenido durante**
Tipo de evento	Compilación
Nombre de clase	Compilación
Nombre del método	Compilación
Argumentos pasados a un método	Compilación
Número de línea en el código fuente de cls	Runtime
Número de línea en el código int generado	Runtime
Nombre de usuario	Runtime
Date/Time	Runtime
Mensaje	Runtime
direccion IP	Runtime

Vamos a crear una clase App.Log que contenga las propiedades de la tabla anterior. Cuando se crea un objeto App.Log, las propiedades de UserName, TimeStamp y ClientIPAddress se completan automáticamente.

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 %String(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()
}
}
```

Macros de registro

Por lo general, las macros se almacenan en archivos * .inc separados que contienen sus definiciones. Los archivos necesarios se pueden incluir en clases usando el comando Include MacroFileName, que en este caso se verá de la siguiente manera: Include App.LogMacro.

Para comenzar, definamos la macro principal que el usuario agregará al código de su aplicación:

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

Esta macro acepta dos argumentos de entrada: Tipo de evento y Mensaje. El usuario define el argumento

Mensaje, pero el parámetro Tipo de evento requerirá macros adicionales con diferentes nombres que identificarán automáticamente el tipo de evento:

```
#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)
```

Por lo tanto, para realizar el registro, el usuario solo necesita colocar la macro $$$LogError("Additional message") en el código de la aplicación.
Todo lo que necesitamos hacer ahora es definir las macros $$$CurrentClass, $$$CurrentMethod, $$$StackPlace, $$$MethodArguments.Comencemos con las tres primeras:

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

%classname, %methodname
las variables se describen en la documentacion. La función $stack devuelve el número de línea del código INT. Para convertirlo en un número de línea CLS, podemos usar este código.

Usemos el paquete %Dictionary para obtener una lista de argumentos de métodos y sus valores. Contiene toda la información sobre las clases, incluidas las descripciones de los métodos. Estamos particularmente interesados en la clase %Dictionary.CompiledMethod y su propiedad FormalSpecParsed, que es una lista:

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

correspondiente a la firma del método. Por ejemplo:

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

tendrá el siguiente valor FormalSpecParsed:

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

Necesitamos que la macro $$$MethodArguments se expanda en el siguiente código (para el método Test):

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

Para lograr esto, tenemos que hacer lo siguiente durante la compilación:

1. Obtenga un nombre de clase y un nombre de método
2. Abra una instancia correspondiente de la clase %Dictionary.CompiledMethod y obtenga su propiedad FormalSpec
3. Conviértalo en una línea de código fuente

Agreguemos los métodos correspondientes a la clase App.Log:

```
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
}
```

Ahora definamos la macro $$$MethodArguments como:

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

Caso de uso

A continuación, creemos una clase App.Use con un método de prueba para demostrar las capacidades del sistema de registro:

```
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")
}
}
```

Como resultado, la macro $$$LogWarn("Text") en el código int se convierte en la siguiente línea:

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

La ejecución de este código creará un nuevo registro de App.Log:

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

Mejoras

Después de haber creado un sistema de registro, aquí hay algunas ideas de mejora:

- En primer lugar, existe la posibilidad de procesar argumentos de tipo objeto ya que nuestra implementación actual solo registra objetos oref.
- Segundo, una llamada para restaurar el contexto de un método a partir de valores de argumentos almacenados.

Procesamiento de argumentos de tipo objeto

La línea que pone un valor de argumento en el registro se genera en el método ArgumentsListToString y tiene este aspecto:

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

Realicemos una refactorización y muévala a un método GetArgumentValue separado que acepte un nombre y clase de variable (todo lo que sabemos de FormalSpecParsed) y genere un código que convertirá la variable en una línea. Usaremos el código existente para los tipos de datos, y los objetos se convertirán en JSON con la ayuda de los métodos SerializeObject (para llamar desde el código de usuario) y WriteJSONFromObject (para convertir un objeto en JSON):

```
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
}
```

Una entrada de registro con un argumento de tipo objeto se ve así:

Restaurando el contexto

La idea de este método es hacer que todos los argumentos estén disponibles en el contexto actual (principalmente en la terminal, para la depuración). Para este fin, podemos usar el parámetro del método ProcedureBlock. Cuando se establece en 0, todas las variables declaradas dentro de dicho método permanecerán disponibles al salir del método. Nuestro método abrirá un objeto de la clase App.Log y deserializará la propiedad Argumentos.

```
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
}
```

Así es como se ve en la 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 = ""
```

¿Que sigue?

La mejora potencial clave es agregar otro argumento a la clase de registro con una lista arbitraria de variables creadas dentro del método.

Conclusiones

Las macros pueden ser bastante útiles para el desarrollo de aplicaciones.

Preguntas

¿Hay alguna manera de obtener el número de línea durante la compilación?

Links

- Part I. Macros
- GitHub repository

#Compilador #Mejores prácticas #Modelo de datos de objetos #Caché

URL de fuente:https://es.community.intersystems.com/post/logging-usando-macros-en-intersystems-cach%C3%A9