記事
Toshihiko Minamoto · 2021年10月12日 11m read

InterSystems IRISでマクロを使用したロギングシステム

前の記事
では、マクロの潜在的なユースケースををレビューしました。そこで、マクロの使用方法についてより包括的な例を見てみることにしましょう。 この記事では、ロギングシステムを設計して構築します。

ロギングシステム

ロギングシステムは、アプリケーションの作業を監視するための便利なツールで、デバッグや監視にかける時間を大幅に節約してくれます。 これから構築するシステムは2つの部分で構成されます。

- ストレージクラス（レコードをログ記録するためのクラス）
- 新しいレコードをログに自動的に追加する一連のマクロ

ストレージクラス

保存する必要のあるもののテーブルを作成し、コンパイル中やランタイム時に、このデータを取得できるタイミングを指定しましょう。 これは、システムの2つ目の部分であるマクロで作業するときに必要となります。そこでは、コンパイル中にできるだけ多くの記録可能な詳細を取得することを目指します。

情報	取得タイミング
イベントタイプ	コンパイル
クラス名	コンパイル
メソッド名	コンパイル
メソッドに渡される引数	コンパイル
clsソースコードの行番号	ランタイム
生成されたintコードの行番号	ランタイム
ユーザー名	ランタイム
日付/時刻	ランタイム
メッセージ	ランタイム
IPアドレス	ランタイム

上記のテーブルのプロパティを含むApp.Logクラスを作成しましょう。 App.Logオブジェクトが作成されると、ユーザー名、日付/時刻、およびIPアドレスプロパティは、自動的に入力されます。

App.Logクラス:

```
Class App.Log Extends %Persistent
{

/// ????????
Property EventType As %String(MAXLEN = 10, VALUELIST = ",NONE,FATAL,ERROR,WARN,INFO,S
TAT,DEBUG,RAW") [ InitialExpression = "INFO" ];

/// ????????????????
Property ClassName As %Dictionary.Classname(MAXLEN = 256);

/// ????????????????
Property MethodName As %String(MAXLEN = 128);

/// int?????
```

```
Property Source As %String(MAXLEN = 2000);

/// cls?????
Property SourceCLS As %String(MAXLEN = 2000);

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

/// ?????????????
Property Arguments As %String(MAXLEN = 32000, TRUNCATE = 1);

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

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

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

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

ロギングマクロ

通常、マクロは、その定義を含む個別の *.incファイルに保存されます。 必要なファイルは、Include MacroFileNameコマンドを使って、クラスに含めることができます。この場合、Include App.LogMacroとなります。

初めに、ユーザーがアプリケーションコードに追加するメインのマクロを定義しましょう。

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

このマクロは、イベントタイプとメッセージの2つの入力引数を受け入れます。 メッセージ引数はユーザーが定義しますが、イベントタイプパラメーターには、イベントタイプを自動的に識別する、別の名前による追加のマクロが必要となります。

```
#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)
```

したがって、ロギングを実行するには、ユーザーはアプリケーションコードに$$$LogError("Additional message")のみを配置するだけで済みます。
後は、$$$CurrentClass、$$$CurrentMethod、$$$StackPlace、$$$MethodArgumentsマクロを定義するのみです。 では、最初の3つから始めましょう。

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

%classnameと%methodname変数は、ドキュメントに記載されています。 $stack
関数はINTコードの行番号を返します。 これをCLS行番号に変換するには、このコードを使用できます。

%Dictionaryパッケージを使用して、メソッド引数とその値のリストを取得しましょう。
これにはメソッドの説明を含む。クラスに関するすべての情報が含まれています。
特に関心があるのは%Dictionary.CompiledMethodクラスとFormalSpecParsedプロパティで、これはリストです。

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

これはメソッドのシグネチャに対応しています。 たとえば次のコードがあるとします。

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

このコードには、次のFormalSpecParsed値があります。

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

$$$MethodArgumentsマクロを次のコードに展開する必要があります（Testメソッド）。

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

これを行うには、コンパイル中に次のことを行う必要があります。

クラス名とメソッド名を取得する
%Dictionary.CompiledMethodクラスの対応するインスタンスを開いて、そのFormalSpecプロパティを取得する
それをソースコード行に変換する
対応するメソッドをApp.Logクラスに追加しましょう。

```
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
}
```

次に$$$MethodArgumentsマクロを以下のように定義しましょう。

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

ユースケース

それでは、ロギングシステムの機能を示すために、Testメソッドを使ってApp.Useクラスを作成しましょう。

```
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")
}
}
```

上記の結果、intコードの$$LogWarn("Text")マクロは次の行に変換されます。

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

このコードを実行すると、新しいApp.Logレコードが作成されます。

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

改善点

ロギングシステムを作成したところで、次のような改善のアイデアがあります。

まず、オブジェクト型引数を処理するようにすることができます。現在の実装では、オブジェクトorefしか記録しないためです。
次に、呼び出しで、保存された引数値からメソッドのコンテキストを復元するようにできます。

オブジェクト型引数の処理

引数の値をログに入れる行は、ArgumentsListToStringメソッドに生成され、次のようになります。

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

リファクタリングを行って、それを、変数名とクラス（FormalSpecParsedから知ることができます）を受け入れる別のGetArgumentValueメソッドに移動し、変数を行に変換するコードを出力してみましょう。 データ型には既存のコードを使用し、オブジェクトは、SerializeObject（ユーザーコードから呼び出すため）とWriteJSONFromObject（オブジェクトをJSONに変換するため）メソッドを使ってJSONに変換されます。

```
ClassMethod GetArgumentValue(Name As %String, ClassName As %Dictionary.CacheClassname
) As %String
{
    If $ClassMethod(ClassName, "%Extends", "%RegisteredObject") {
        // ????????
        Return "_##class(App.Log).SerializeObject("_Name _ ")_"
    } Else {
        // ??????
        Return "_$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=""

        //IO???????????????????????????????
        Use $io::("^"_$ZNAME)

        //?????????????
        Do ##class(%Device).ReDirectIO(1)

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

```
    //????????/??????????????????
    If (OldMnemonic '= "") {
        Use OldIO::("^"_OldMnemonic)
    } Else {
        Use OldIO
    }
    Do ##class(%Device).ReDirectIO(OldIORedirected)

    Quit Str

    // IO????????????
    // ??????????????????????
rchr(c)      Quit
    // ??????????????????????
rstr(sz,to)  Quit
    // ??????????????????
wchr(s)      Do output($char(s))  Quit
    // ??????????????????????????
wff()        Do output($char(12))  Quit
    // ??????????????????
wnl()        Do output($char(13,10))  Quit
    // ??????????????????
wstr(s)      Do output(s)  Quit
    // ??????????????????
wtab(s)      Do output($char(9))  Quit
    // ??????????????????????????
    // ????Str???????
output(s)    Set Str = Str_s Quit
}
```

オブジェクト型引数のログエントリは次のようになります。

コンテキストの復元

このメソッドの主旨は、すべての引数を現在のコンテキストで（主にデバッグ用のターミナルで）使用できるようにすることです。 これを行うために、ProcedureBlockメソッドパラメーターを使用することができます。 0に設定すると、そのようなメソッドに宣言されたすべての変数はメソッドを終了してもそのまま使用することができます。
ここでのメソッドは、App.Logクラスのオブジェクトを開いて、Argumentsプロパティを逆シリアル化します。

```
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") {
```

```
        // ????????
        Set st = ##class(%ZEN.Auxiliary.jsonProvider).%ConvertJSONToObject(String,,.o
bj)
        Return:$$$ISOK(st) obj
    }
    Return String
}
```

ターミナルでは次のように表示されます。

```
> 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 = ""
```

この続きは？

鍵となる潜在的な改善点は、メソッド内に作成された任意の変数リストを使用して、ログクラスに別の引数を追加することです。

まとめ

マクロはアプリケーション開発に非常に役立ちます。

質問

コンパイル中に行番号を取得する方法はありますか？

リンク

- パート1. マクロ
- GitHubリポジトリ

#オブジェクトデータモデル #コンパイラ #Caché #InterSystems IRIS

ソースURL:https://jp.community.intersystems.com/post/intersystems-iris%E3%81%A7%E3%83%9E%E3%82%AF%E3%83%AD%E3%82%92%E4%BD%BF%E7%94%A8%E3%81%97%E3%81%9F%E3%83%AD%E3%82%AE%E3%83%B3%E3%82%B0%E3%82%B7%E3%82%B9%E3%83%86%E3%83%A0