Article
Timur Safin · Jul 18, 2016 15m read

Remote proxy objects via dynamic dispatch

This article created as side effect of preparations to the longer set of articles about simple, but still handy MapReduce implementation in Caché. I was looking for relatively easy way to pass arguments to (potentially) multiple targets via remote calling facilities. And after several attempts I have realized that we do have very powerful mechanism in the Caché ObjectScript which might be of particular help here – dynamic dispatch for methods and properties.

Since Caché 5.2 there are multiple predefined methods inherited by any object based on %RegisteredObject (at the moment they are established in %Library.SystemBase) and which are called if there is unknown method or property call, for the name not defined in the class metadata.

Method %DispatchMethod (Method As %String, Args...)	Unknown method call or access to unknown multidimensional property (thei identical)
ClassMethod %DispatchClassMethod (Class As %String, Method As %String, Args...)	Unknown classmethod call for the given class
Method %DispatchGetProperty (Property As %String)	Getter for access to an unknown property
Method %DispatchSetProperty (Property As %String, Val)	Setter for an unknown property
Method %DispatchSetMultidimProperty (Property As %String, Val, Subs...)	Setter for an unknown multidimensional property (*not used in this story*)
Method %DispatchGetModified (Property As %String)	Get “ modified ” flag value for an unknown property (*not used in this story*)
Method %DispatchSetModified (Property As %String, Val)	Set “ modified ” flag value for an unknown property (*not used in this story*)

For simplicity sake, we will use only unknown method calls and access to unknown properties, but for the product grade systems, you might eventually override all mentioned methods.

Logging proxy sample

There is good example of dynamic dispatch methods usage in the CACHELIB, and it is called %ZEN.proxyObject, historically it allowed to operate dynamic properties, even at the times when there was no DocumentDB API and there was no native JSON support in the kernel.

While still approaching our long-term goal (remote proxy object implementation), let ’ s experiment with something simpler, i.e. to create “ logging proxy ” , where we will wrap any access to the particular object API using dynamic dispatch methods, while logging each event. [Actually, this provides side effect similar to mocking techniques used in other language environments].

Let assume we have rudimentary Person class known as Sample.SimplePerson (due to some rare incident, which

is very similar to Sample.Person from the SAMPLES namespace J)

```
DEVLATEST:15:23:32:MAPREDUCE>set p =
##class(Sample.SimplePerson).%OpenId(2)

DEVLATEST:15:23:34:MAPREDUCE>zw p
p=<OBJECT REFERENCE>[1@Sample.SimplePerson]
+----------------- general information ---------------
|      oref value: 1
|      class name: Sample.SimplePerson
|           %%OID: $lb("2","Sample.SimplePerson")
| reference count: 2
+----------------- attribute values ------------------
|       %Concurrency = 1  <Set>
|                Age = 9
|           Contacts = 23
|               Name = "Waal,Nataliya Q."
+-----------------------------------------------------
```

Let ' s wrap access to this object properties to the logging class instance, and each access to property or method call will write to the logging global somewhere.

```
/// simple sample of a logging proxy object:
/// each access (via call or property access) will
/// be logged to the designated global
Class Sample.LoggingProxy Extends %RegisteredObject
{
/// log access log to this global
Parameter LoggingGlobal As %String = "^Sample.LoggingProxy";
/// keep openedobject for the proxy access later
Property OpenedObject As %RegisteredObject;

/// generic log utility, which save new string as a next global entry
ClassMethod Log(Value As %String)
{
    #dim gloRef = ..#LoggingGlobal
    set @gloRef@($increment(@gloRef)) = Value
}

/// more convenient log method for writing prefix (i.e. method name)
/// and arguments we were called in
ClassMethod LogArgs(prefix As %String, args...)
{
    #dim S as %String = $get(prefix) _ ": " _ $get(args(1))
    #dim i as %Integer
    for i=2:1:$get(args) {
        set S = S_","_args(i)
    }
    do ..Log(S)
}

/// open instance of a different class using given %ID
ClassMethod %CreateInstance(className As %String, %ID As %String) As Sample.LoggingPr
```

```
oxy
{
    #dim wrapper = ..%New()
    set wrapper.OpenedObject = $classmethod(className, "%OpenId", %ID)
    return wrapper
}

/// log arguments and then dispatch dynamically method to the proxy object
Method %DispatchMethod(methodName As %String, args...)
{
    do ..LogArgs(methodName, args...)
    return $method(..OpenedObject, methodName, args...)
}

/// log arguments and then dispatch dynamically property access to the proxy object
Method %DispatchGetProperty(Property As %String)
{
    #dim Value as %String = $property(..OpenedObject, Property)
    do ..LogArgs(Property, Value)
    return Value
}

/// log arguments and then dispatch dynamically property access to the proxy object
Method %DispatchSetProperty(Property, Value As %String)
{
    do ..LogArgs(Property, Value)
    set $property(..OpenedObject, Property) = Value
}

}
```

1. There is class parameter #LoggingGlobal which defines where we store our log (^Sample.LogginGlobal in this case);
2. There are simple Log(Arg) and LogArgs(prefix, args…) methods which allow conveniently write to this global passed argument or list of them;
3. %DispatchMethod, %DispatchGetPRoperty or %DispatchSetProperty handle their respective parts of unknown method call or unknown property access. They log each access event via LogArgs method, then directly call the wrapped object (..%OpenedObject) methods or access properties;
4. And there is the “ factory method ” %CreateInstance which is opening instance of an asked classname given their instance %ID. The created object is “ wrapped ” to the Sample.LogginProxy object, reference to which is returned from this classmethod.

There is nothing super fancy so far, only 70 lines of simple Caché ObjectScript code, which introduce an idiom of method/property calls with side-effect. Please see how it works in real life:

```
DEVLATEST:15:25:11:MAPREDUCE>set w = ##class(Sample.LoggingProxy).%CreateInstance("Sa
mple.SimplePerson", 2)

DEVLATEST:15:25:32:MAPREDUCE>zw w
w=<OBJECT REFERENCE>[1@Sample.LoggingProxy]
+----------------- general information ---------------
|      oref value: 1
|      class name: Sample.LoggingProxy
| reference count: 2
+----------------- attribute values ------------------
```

```
|           (none)
+----------------- swizzled references ---------------
|     i%OpenedObject = ""
|     r%OpenedObject = 2@Sample.SimplePerson
+-----------------------------------------------------

DEVLATEST:15:25:34:MAPREDUCE>w w.Age
9
DEVLATEST:15:25:41:MAPREDUCE>w w.Contacts
23
DEVLATEST:15:25:49:MAPREDUCE>w w.Name
Waal,Nataliya Q.
DEVLATEST:15:26:16:MAPREDUCE>zw ^Sample.LoggingProxy
^Sample.LoggingProxy=4
^Sample.LoggingProxy(1)="Age: 9"
^Sample.LoggingProxy(2)="Contacts: 23"
^Sample.LoggingProxy(3)="Name: Waal,Nataliya Q."
```

If you compare results to the direct access of an instance of Sample.SimplePerson above then you ' ll see that results are quite consistent and as expected.

Remote proxy

Careful reader should still remember, all this stuff we needed just to have easier way for remote proxy objects. But what was wrong with the normal way, using %Net.RemoteConnection?

Many things (though " deprecated " status of this class is not in this list J).

%Net.RemoteConnection is using c-binding facilities (which are wrapper around cpp-binding service) for calling methods from remote Caché instances. If you know its address, target namespace, and you are able to login then you have pretty much all to establish remote procedure call to this Caché node. The problem with this API – it ' s not the easiest and, certainly, not the least verbose way to use:

```
Class MR.Sample.TestRemoteConnection Extends %RegisteredObject
{

ClassMethod TestMethod(Arg As %String) As %String
{
    quit $zu(5)_"^"_##class(%SYS.System).GetInstanceName()_"^"_$i(^MR.Sample.TestRemo
teConnectionD)
}

ClassMethod TestLocal()
{
    #dim connection As %Net.RemoteConnection = ##class(%Net.RemoteConnection).%New()
    #dim status As %Status = connection.Connect("127.0.0.1",$zu(5),^%SYS("SSPort"),"_
SYSTEM","SYS")
    set status = connection.ResetArguments()
    set status = connection.AddArgument("Hello", 0 /*by ref*/, $$$cbindStringId)
    #dim rVal As %String = ""
    set status = connection.InvokeClassMethod(..%ClassName(1), "TestMethod", .rVal, 1
 /*has return*/, $$$cbindStringId)
    zw rVal
    do connection.Disconnect()
}
```

…

}

i.e. after establishing connection, and before calling classmethod or object instance method, you supposed to prepare a list of arguments of this call (starting from ResetArguments, and then adding next argument via AddArgument, which in turn should take care about proper cpp-binding type information of an argument, its in/out direction, and many other things).

Also, for me, it was quite annoying to not have simpler way to retrieve return value from other side, because all invocations will give me just execution status code instead, and passing returning value elsewhere in the argument list of the method.

I am too old and am too lazy for such boring games!

I want to have less verbose, more convenient way to pass argument to the functions. Remember, we have args… facilities in the language syntax (for passing variable number of arguments to the function), why simply not to use it for wrapping all the dirty details?

```
/// sample of a remote proxy using %Net.RemoteConnection
Class Sample.RemoteProxy Extends %RegisteredObject
{
Property RemoteConnection As %Net.RemoteConnection [ Internal ];
Property LastStatus As %Status [ InitialExpression = {$$$OK} ];

Method %OnNew() As %Status
{
    set ..RemoteConnection = ##class(%Net.RemoteConnection).%New()

    return $$$OK
}

/// create new instance of a given class name
Method %CreateInstance(className As %String) As Sample.RemoteProxy.Object
{
    #dim instanceProxy As Sample.RemoteProxy.Object = ##class(Sample.RemoteProxy.Obje
ct).%New($this)
    return instanceProxy.%CreateInstance(className)
}

Method %OpenObjectId(className As %String, Id As %String) As Sample.RemoteProxy.Objec
t
{
    #dim instanceProxy As Sample.RemoteProxy.Object = ##class(Sample.RemoteProxy.Obje
ct).%New($this)
    return instanceProxy.%OpenObjectId(className, Id)
}

/// pass the configuration object { "IP": IP, "Namespace" : Namespace, ... }
Method %Connect(Config As %Object) As Sample.RemoteProxy
{
    #dim sIP As %String = Config.IP
    #dim sNamespace As %String = Config.Namespace
    #dim sPort As %String = Config.Port
    #dim sUsername As %String = Config.Username
```

```
    #dim sPassword As %String = Config.Password
    #dim sClientIP As %String = Config.ClientIP
    #dim sClientPort As %String = Config.ClientPort

    if sIP = "" { set sIP = "127.0.0.1" }
    if sPort = "" { set sPort = ^%SYS("SSPort") }
    set ..LastStatus = ..RemoteConnection.Connect(sIP, sNamespace, sPort,
                                                  sUsername, sPassword,
                                                  sClientIP, sClientPort)

    return $this
}

ClassMethod ApparentlyClassName(CompoundName As %String, Output ClassName As %String,
 Output MethodName As %String) As %Boolean [ Internal ]
{
    #dim returnValue As %Boolean = 0

    if $length(CompoundName, "::") > 1 {
        set ClassName = $piece(CompoundName, "::", 1)
        set MethodName = $piece(CompoundName, "::", 2, *)

        return 1
    } elseif $length(CompoundName, "'") > 1 {
        set ClassName = $piece(CompoundName, "'", 1)
        set MethodName = $piece(CompoundName, "'", 2, *)

        return 1
    }

    return 0
}

/// log arguments and then dispatch dynamically method to the proxy object
Method %DispatchMethod(methodName As %String, args...)
{
    #dim className as %String = ""

    if ..ApparentlyClassName(methodName, .className, .methodName) {
        return ..InvokeClassMethod(className, methodName, args...)
    }
    return 1
}

Method InvokeClassMethod(ClassName As %String, MethodName As %String, args...)
{
    #dim returnValue = ""
    #dim i as %Integer
    do ..RemoteConnection.ResetArguments()
    for i=1:1:$get(args) {
        set ..LastStatus = ..RemoteConnection.AddArgument(args(i), 0)
    }
    set ..LastStatus = ..RemoteConnection.InvokeClassMethod(ClassName, MethodName, .r
eturnValue, $quit)
    return returnValue
}

}
```

1st simplification which I ' ve introduced here – I do not use regular arguments in the %Connect method, but rather pass dynamic JSON object for configuration. Consider this as syntaxic sugar, and use as named argument idiom in other languages (where named-arguments are usually implemented the similar way, via on-the-fly construction of key-value pairs for the hash-object passed to the function):

```
DEVLATEST:16:27:18:MAPREDUCE>set w = ##class(Sample.RemoteProxy).%New().%Connect({"Na
mespace":"SAMPLES",?
 "Username":"_SYSTEM", "Password":"SYS"})

DEVLATEST:16:27:39:MAPREDUCE>zw w
w=<OBJECT REFERENCE>[1@Sample.RemoteProxy]
+----------------- general information ---------------
|      oref value: 1
|      class name: Sample.RemoteProxy
| reference count: 2
+----------------- attribute values ------------------
|         LastStatus = 1
+----------------- swizzled references ---------------
|           i%Config = ""
|           r%Config = ""
| i%RemoteConnection = ""
| r%RemoteConnection = 2@%Net.RemoteConnection
+-----------------------------------------------------
```

There is yet another idiom used – wherever possible, I return $this instance reference as returning value, thus it becomes possible to apply " chaining " for method calls. Which is also helping us to reduce a number of code we have to write. [Yes, I ' m that lazy!]

Class method call problem

This root %Net.RemoteConnection wrapper object could not do much, if there is no place in the context to store references to the created instances. [We will address this problem later, in the different class] The only thing we could do now – is somehow simplify class-method calls, which might be called without object context. We could redefine %DispatchClassMethod here, but it will not help us if we want to have generic remote proxy wrapper [and we do want], which will serve us for any remote class. [Though, it may help if you have 1:1 relationship and some specialized local class as wrapper for some particular remote class] Thus, for the generic case like here, we need something different, but, hopefully, still be as much convenient as possible. InvokeClassMethod implementation above is generally ok, but not as handy as it might be possible. We will try to introduce something more elegant soon.

But, before then, let us look in what could be written as a method or property identifier. It ' s not well known fact (at least is not very spread inside of English-based community) that ObjectScript method names could be anything what is considered alphanumeric symbols for the given Caché locale (i.e. it could be not only A-Za-z0-9 for Llatin-based locale, but also could be any other " alphabetic " symbol, like А-Яа-я for Russian locales[see this StackOverflow discussion] We could even use emoji symbols as inner identifier separator, if we would manage to create such Caché locale. However, in general, any locale specific trick would not fly well, as a generic solution, so we not spend much time here.

On the other hand, the idea of using special separator inside of method name looks fruitful. We may handle separator in the %DispatchMethod call, which then will extract class-name encoded and will dispatch to the corresponding class-method function elsewhere.

So returning to the syntax of allowed method names, it is even less known fact that you could put pretty much

everything to the method or property name if you properly quote them. For example, I want to call LogicalToDisplay classmethod in the class Cinema.Duration. The syntax would be:

```
DEVLATEST:16:27:41:MAPREDUCE>set w = ##class(Sample.RemoteProxy).%New().%Connect({"Na
mespace":"SAMPLES",?
 "Username":"_SYSTEM", "Password":"SYS"})

DEVLATEST:16:51:39:MAPREDUCE>write w."Cinema.Duration::LogicalToDisplay"(200)
3h20m
```

Simple and elegant, isn ' t it?

[This special processing of a method name is done in the ApparentlyClassName function, where we are looking for " :: " (double colon) or " ' " (single quote) as separator between classname and methodname.]

But please take into consideration, that if you want to call class-method which outputs something to the screen then you are out of luck: cpp-binding protocol will not transfer you back output to the screen, it ' s not redirected. It returns values, not side-effects.

```
DEVLATEST:16:51:47:MAPREDUCE>do w."Sample.Person::PrintPersons"(1)
```

Remote instances proxies

As you might recognize in the Sample.RemoteProxy above, we didn ' t do much there, only establish connection and call class methods. However, for creation of a remote instance wrapper (%CreateInstance) and for opening remote instance by %ID (%OpenObjectId) we use another class facilities – that is responsibility of a %Sample.RemoteProxy.Object wrapper class.

```
Class Sample.RemoteProxy.Object Extends %RegisteredObject
{
/// keep openedobject for the proxy access later
Property OpenedObject As %Binary;
Property Owner As Sample.RemoteProxy [ Internal ];
Property LastStatus As %Status [ InitialExpression = {$$$OK}, Internal ];

Method RemoteConnection() As %Net.RemoteConnection [ CodeMode = expression ]
{
..Owner.RemoteConnection
}

Method %OnNew(owner As Sample.RemoteProxy) As %Status
{
    set ..Owner = owner
    return $$$OK
}

/// open instance of a different class using given %ID
Method %CreateInstance(className As %String) As Sample.RemoteProxy.Object
{
```

```
    #dim pObject As %RegisteredObject = ""
    set ..LastStatus = ..RemoteConnection().CreateInstance(className, .pObject)
    set ..OpenedObject = ""
    if $$$ISOK(..LastStatus) {
        set ..OpenedObject = pObject
    }
    return $this
}

/// open instance of a different class using given %ID
Method %OpenObjectId(className As %String, Id As %String) As Sample.RemoteProxy.Objec
t
{
    #dim pObject As %RegisteredObject = ""
    set ..LastStatus = ..RemoteConnection().OpenObjectId(className, Id, .pObject)
    set ..OpenedObject = ""
    if $$$ISOK(..LastStatus) {
        set ..OpenedObject = pObject
    }
    return $this
}

Method InvokeMethod(MethodName As %String, args...) [ Internal ]
{
    #dim returnValue = ""
    #dim i as %Integer
    #dim remoteConnection = ..RemoteConnection()
    do remoteConnection.ResetArguments()
    for i=1:1:$get(args) {
        set ..LastStatus = remoteConnection.AddArgument(args(i), 0)
    }
    set ..LastStatus = remoteConnection.InvokeInstanceMethod(..OpenedObject, MethodNa
me, .returnValue, $quit)
    return returnValue
}

/// log arguments and then dispatch dynamically method to the proxy object
Method %DispatchMethod(methodName As %String, args...)
{
    //do ..LogArgs(methodName, args...)
    return ..InvokeMethod(methodName, args...)
}

/// log arguments and then dispatch dynamically property access to the proxy object
Method %DispatchGetProperty(Property As %String)
{
    #dim value = ""
    set ..LastStatus = ..RemoteConnection().GetProperty(..OpenedObject, Property, .va
lue)
    return value
}

/// log arguments and then dispatch dynamically property access to the proxy object
Method %DispatchSetProperty(Property, Value As %String) As %Status
{
   set ..LastStatus = ..RemoteConnection().SetProperty(..OpenedObject, Property, Valu
e)
    return ..LastStatus
}
```

```
}
```

There is common connection object, which Sample.RemoteProxy used, but many remote object instances Sample.RemoteProxy.Object, each of which should get access to the remote connection via their ..Owner reference passed at the initialization time (see %OnNew argument passed).

There is relatively convenient InvokeMethod created, which handles method calls with any number of arguments, and which marshalls arguments to the corresponding %Net.RemoteConnection calls (i.e. call ResetArguments and many AddArgument), and which then calls to %NetRemoteConnection::InvokeInstanceMethod for actual method execution, processing their return value.

```
DEVLATEST:19:23:54:MAPREDUCE>set w = ##class(Sample.RemoteProxy).%New().%Connect({"Na
mespace":"SAMPLES", ?
"Username":"_SYSTEM", "Password":"SYS"})
…
DEVLATEST:19:23:56:MAPREDUCE>set p = w.%OpenObjectId("Sample.Person",1)

DEVLATEST:19:24:05:MAPREDUCE>write p.Name
Quince,Maria B.
DEVLATEST:19:24:11:MAPREDUCE>write p.SSN
369-27-1697
DEVLATEST:19:24:17:MAPREDUCE>write p.Addition(1,2)
3
```

In this code above, we instantiate remote proxy to the “ Sample.Person ” instance from “ SAMPLES ” namespace. And then call its method(s) or access properties. Quite simple again?

Kind of conclusion

This is not yet product quality code:

- there is no proper handling of errors (they should be raising exceptions if there is something wrong happen),
- there is no handling of disconnect or graceful shutdown of a whole set of instantiated objects
- but even today, in their current state, this set of classes show how remote proxy could be used in the

  readable and maintainable code. Which worth all efforts.

All the code mentioned in the article is available via this gist.

#Code Snippet #Object Data Model #Caché