Artículo
Mathew Lambert · 25 nov, 2020 Lectura de 17 min

Objetos de proxy remoto mediante despacho dinámico

Este artículo fue creado como efecto secundario de las preparaciones para un conjunto más extendido de artículos sobre la simple, pero útil, implementación de MapReduce en Caché. Estaba buscando una forma relativamente fácil de pasar argumentos a (potencialmente) múltiples objetivos mediante mecanismos de invocación remota. Y tras varios intentos, me di cuenta de que en el ObjectScript de Caché contamos con mecanismos muy potentes que podrían ser de utilidad para esto: dynamix dispatching para métodos y propiedades.

A partir de Caché 5.2 existen múltiples métodos predefinidos heredados por cualquier objeto basado en %RegisteredObject (en el momento en que se establecen en %Library.SystemBase) y los cuales se llaman si existe un método desconocido o una llamada a una propiedad, para el nombre no definido en los metadatos de la clase.

Method %DispatchMethod (Method As %String, Args...)	Llamada a método desconocido o acceso a propiedad multidimensional des sintaxis es idéntica)
ClassMethod %DispatchClassMethod (Class As %String, Method As %String, Args...)	Llamada a método de clase desconocido para la clase dada
Method %DispatchGetProperty (Property As %String)	Getter para acceso a una propiedad desconocida
Method %DispatchSetProperty (Property As %String, Val)	Setter para una propiedad desconocida
Method %DispatchSetMultidimProperty (Property As %String, Val, Subs...)	Setter para una propiedad multidimensional desconocida (*no usado en esta historia)*
Method %DispatchGetModified (Property As %String)	Recuperar valor de la marca "modificado" para una propiedad desconocida *(no usado en esta historia)*
Method %DispatchSetModified (Property As %String, Val)	Setear valor de la marca "modificado" para una propiedad desconocida *(no usado en esta historia)*

Para simplificar, solo usaremos las llamadas a métodos desconocidos y el acceso a propiedades desconocidas, pero para los sistemas que tengan un grado de producto se podrían sobreescribir todos los métodos mencionados.

Ejemplo de proxy de registro

Existe un buen ejemplo de uso de métodos de dynamic dispatching en CACHELIB, que se llama %ZEN.proxyObject. Históricamente permitía operar propiedades dinámicas, incluso en momentos en los que no existía la API DocumentDB y en el kernel no había soporte nativo para JSON.

Mientras nos aproximamos a nuestro objetivo de largo plazo (implementación de un objeto proxy remoto), experimentemos con algo más simple. Es decir, creemos un "proxy de registro", donde usaremos métodos de dynamic dispatching para wrapear cualquier acceso a la API del objeto particular, a la vez que registramos cada evento. [En realidad, esto logra un efecto secundario similar a las técnicas de mocking usadas en otros entornos de lenguajes].

Supongamos que tenemos una clase Person rudimentaria, conocida como Sample. SimplePerson (debido a alguna rara coincidencia, lo que es muy similar a Sample.Person para el namespace J de SAMPLES).

```
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."
+-----------------------------------------------------
```

Vamos a wrapear el acceso a las propiedades de este objeto con la instancia de clase de registro, y cada acceso a una llamada de propiedad o método escribirá al global de registro en algún sitio.

```
/// ejemplo simple de un objeto de proxy de registro:
/// cada acceso (mediante llamada o acceso a propiedad)
/// se registrará en el global designado
Class Sample.LoggingProxy Extends %RegisteredObject
{
/// registrar el registro de acceso a este global
Parameter LoggingGlobal As %String = "^Sample.LoggingProxy";
/// mantener openedobject para el posterior acceso al proxy
Property OpenedObject As %RegisteredObject;

/// utilidad de registro genérico, que guardará la nueva cadena de texto como una sig
uiente entrada global
ClassMethod Log(Value As %String)
{
    #dim gloRef = ..#LoggingGlobal
    set @gloRef@($increment(@gloRef)) = Value
}

/// método de registro más conveniente para escribir un prefijo (es decir, nombre de
método)
/// y los argumentos en los que nos llamaron
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)
```

```
}

/// abrir una instancia de una clase diferente usando un %ID dado
ClassMethod %CreateInstance(className As %String, %ID As %String) As Sample.LoggingPr
oxy
{
    #dim wrapper = ..%New()
    set wrapper.OpenedObject = $classmethod(className, "%OpenId", %ID)
    return wrapper
}

/// registrar argumentos y luego dispatching dinámico al objeto proxy
Method %DispatchMethod(methodName As %String, args...)
{
    do ..LogArgs(methodName, args...)
    return $method(..OpenedObject, methodName, args...)
}

/// registrar argumentos y luego despachar el acceso a la propiedad dinámicamente al
objeto proxy
Method %DispatchGetProperty(Property As %String)
{
    #dim Value as %String = $property(..OpenedObject, Property)
    do ..LogArgs(Property, Value)
    return Value
}

/// registrar argumentos y luego despachar el acceso a la propiedad dinámicamente al
objeto proxy
Method %DispatchSetProperty(Property, Value As %String)
{
    do ..LogArgs(Property, Value)
    set $property(..OpenedObject, Property) = Value
}

}
```

1. Existe un parámetro de clase #LoggingGlobal que define dónde guardamos nuestro registro (^Sample.LogginGlobal en este caso);
2. Hay métodos Log(Arg) y LogArgs(prefijo, argumentos⋯) simples que permiten escribir convenientemente a este argumento global pasado o a una lista de ellos;
3. %DispatchMethod, %DispatchGetPRoperty o %DispatchSetProperty manejan sus partes respectivas de llamadas a métodos desconocidos o accesos a propiedades desconocidas. También registran cada evento de acceso mediante el método LogArgs y luego llaman directamente a los métodos del objeto envuelto (..%OpenedObject) o acceden a sus propiedades;
4. Y también está el "método factory" %CreateInstance, que abre una instancia de un nombre de clase solicitado dado su ID de instancia. El objeto creado se "wrapea" sobre el objeto Sample.LogginProxy, y se devuelve una referencia al mismo desde este método de clase.

Hasta ahora no hay nada muy avanzado, apenas 70 líneas de código Caché ObjectScript simples, que introducen una expresión de llamadas a métodos/propiedades con efecto secundario. Veamos cómo funciona en la vida real:

```
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."
```

Si se comparan los resultados con el acceso directo de una instancia en Sample.SimplePerson arriba, se ve que los resultados son bastante consistentes, como es de esperar.

Proxy remoto

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?

El lector más detallista debería recordar, necesitamos todo esto para tener una forma mas sencilla de proxy objects. ¿Pero qué tenía de malo la forma normal, usando %Net.RemoteConnection?

Muchas cosas (aunque el estado "obsoleto" de esta clase no está en esta lista).

%Net.RemoteConnection usa funcionalidades de c-binding (que se envuelven alrededor del servicio cpp-binding) para llamar a métodos desde instancias remotas de Caché. Si conoces su dirección, su namespace objetivo, y puedes iniciar sesión, entonces tienes todo lo necesario para establecer una llamada de procedimiento remoto a este nodo Caché. El problema con esta API es que no es la forma más sencilla ni menos minuciosa de usar:

```
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()
}

…

}
```

Por ejemplo, después de establecer la conexión, y antes de llamar al método de clase o al método de instancia de objeto, se deberá preparar una lista de argumentos de esta llamada (empezando por los ResetArguments, para agregar luego el siguiente argumento con AddArgument, que a su vez debería encargarse de asegurar la correcta información de tipo de cpp-binding de un argumento, su dirección de entrada/salida y otras cosas).

Además, para mí, era bastante molesto no tener una forma más simple de recuperar el valor de retorno desde el otro lado, ya que todas las invocaciones simplemente me darán el código de estado de ejecución, y pasarán el valor de retorno a alguna otra parte de la lista de argumentos del método.

¡Soy demasiado viejo y vago para juegos tan aburridos!

Quiero tener una forma menos minuciosa y más práctica de pasar argumentos a las funciones. Recuerda que tenemos las facilidades del args··· en la sintaxis del idioma (para pasar un número variable de argumentos a la función), ¿por qué simplemente no usarlo para envolver todos los detalles sucios?

```
/// muestra de un proxy remoto usando %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
}

/// crear nuevas instancias de un nombre de clase dado
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)
}

/// pasar el objeto de configuración { "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
}

/// registrar argumentos y luego despachar el método dinámicamente al objeto proxy
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
}

}
```

La 1ª simplificación que introduzco aquí: no uso argumentos regulares en el método %Connect, sino que paso un objeto JSON dinámico para la configuración. Considera esto como "azúcar" para la sintaxis, y usar como expresión del argumento nombrado en otros lenguajes (en los que los argumentos nombrados usualmente se implementan de forma similar, mediante una construcción informal de pares de clave-valor para el objeto hash pasado a la función):

```
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
+-----------------------------------------------------
```

Queda otra expresión que se usa: siempre que sea posible, devuelvo $this referencia de instancia como valor de devolución, y así es posible aplicar "encadenado" para llamadas a métodos. Y eso también nos ayuda a reducir la cantidad de código que tenemos que escribir. [Sí, así de vago soy!]

Problema de llamar a un método de clase

Este objeto wrapper %Net.RemoteConnection raíz no podía hacer mucho si no hay un lugar en el contexto para almacenar las referencias a la instancias creadas. [Trataremos este problema más tarde, en una clase diferente]. Lo único que podemos hacer ahora es simplificar de alguna forma las llamadas a métodos clase, que se pueden llamar sin contexto de objeto. Aquí podríamos redefinir %DispatchClassMethod, pero no nos ayudará si queremos tener un envoltorio proxy remoto "genérico" [cosa que queremos], que nos servirá para cualquier clase remota. [Si bien podría ayudar si tienes una relación de 1:1 y alguna clase local especializada como envoltorio para alguna clase remota particular]. Así, para un caso genérico como este, necesitamos algo distinto pero, esperemos, igualmente será tan conveniente como sea posible. La anterior implementación de InvokeClassMethod en general es buena, pero no tan práctica como podría ser posible. En breve intentaremos mostrar algo más elegante.

Pero, por ahora, analicemos qué se podría escribir como identificador de método o propiedad. No es algo muy sabido (o al menos no es muy difundido en la comunidad) que los nombres de métodos ObjectScript pueden estar

formados de cualquier combinación de símbolos alfanuméricos para la localización de Caché dada (es decir, no solo puede ser A-Za-z0-9 para localizaciones de alfabeto latino, sino que también puede tener cualquier otro símbolo "alfabético", como А-Яа-я para localizaciones al ruso). [echa un vistazo a esta conversación en StackOverflow]. Incluso podríamos usar emojis como separador de identificador interno, si nos ingeniáramos para crear tal localización de Caché. Sin embargo, en general, cualquier truco específico para una localización no funcionará bien como solución genérica, por lo que no invertiremos mucho tiempo en esto.

Por otro lado, la idea de usar un separador especial dentro del nombre de un método parece útil. Podríamos manejar el separador en la llamada %DispatchMethod, que luego extraerá el nombre de clase cifrado y hará el dispatching a la función del método de clase correspondiente en otro sitio.

Así que, volviendo a la sintaxis de los nombres de métodos permitidos, es aún menos conocido que se puede poner prácticamente *todo* en el nombre del método o propiedad si se cita correctamente. Por ejemplo, quiero llamar al método de clase LogicalToDisplay en la clase Cinema.Duration. La sintaxis sería:

```
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 y elegante, ¿verdad?

[Este procesamiento especial de un nombre de método se hace en la función ApparentlyClassName, en la que buscamos “ :: ” (doble dos puntos) o “ ’ ” (comilla simple) como separador entre el nombre de clase y el nombre de método.]

Pero, por favor, ten en cuenta que si quieres llamar a un método de clase que tiene como salida algo a imprimir en pantalla, entonces no tendrás suerte: el protocolo cpp-binding no transferirá esa salida de vuelta a la pantalla, no se redirecciona. Devuelve valores, no efectos secundarios.

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

Proxies de instancias remotas

Como quizá reconozcas del Sample.RemoteProxy anterior, no hicimos mucho allí, tan solo establecer los métodos de clase de llamada y conexión. Sin embargo, para crear un wrapper de instancia remota (%CreateInstance) y para abrir una instancia remota mediante %ID (%OpenObjectId), usamos otras facilidades de clase (que es responsable de una clase de envoltorio %Sample.RemoteProxy.Object).

```
Class Sample.RemoteProxy.Object Extends %RegisteredObject
{
/// mantener openedobject para el posterior acceso al proxy
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
}

/// abrir una instancia de una clase diferente usando un %ID dado
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
}

/// abrir una instancia de una clase diferente usando un %ID dado
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
}

/// registrar argumentos y luego despachar el método dinámicamente al objeto proxy
Method %DispatchMethod(methodName As %String, args...)
{
    //do ..LogArgs(methodName, args...)
    return ..InvokeMethod(methodName, args...)
}

/// registrar argumentos y luego despachar el acceso a la propiedad dinámicamente al
objeto proxy
Method %DispatchGetProperty(Property As %String)
```

```
{
    #dim value = ""
    set ..LastStatus = ..RemoteConnection().GetProperty(..OpenedObject, Property, .va
lue)
    return value
}

/// registrar argumentos y luego despachar el acceso a la propiedad dinámicamente al
objeto proxy
Method %DispatchSetProperty(Property, Value As %String) As %Status
{
   set ..LastStatus = ..RemoteConnection().SetProperty(..OpenedObject, Property, Valu
e)
    return ..LastStatus
}

}
```

Existe un objeto de conexión común, que era usado por Sample.RemoteProxy, pero muchas instancias de objeto remoto Sample.RemoteProxy.Object, cada una de las cuales debería tener acceso a la conexión remota a través de su referencia de propietario pasada al momento de la inicialización (puedes ver el argumento %OnNew pasado).

Existe un InvokeMethod creado, relativamente conveniente, que maneja las llamadas a métodos con cualquier cantidad de argumentos, y que ordena los argumentos para las llamadas %Net.RemoteConnection correspondientes (es decir, llamar a ResetArguments y muchos AddArgument), y que luego llama a %NetRemoteConnection::InvokeInstanceMethod para la verdadera ejecución del método, para luego procesar su valor de devolución.

```
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
```

En el código anterior, instanciamos el proxy remoto a la instancia de “ Sample.Person ” del namespace “ SAMPLES ”. Y luego llamamos a su(s) método(s) o prioridades de acceso. También es bastante sencillo, ¿verdad?

Conclusión

Esto aún no es código con calidad apta para producto:

- no hay una correcta gestión de errores (deberían generar excepciones si sucede algo incorrecto),
- no hay una gestión de desconexión o apagado ágil de un conjunto completo de objetos instanciados,
- pero incluso hoy, en su estado actual, este conjunto de clases muestra cómo se podría usar un proxy

remoto en el código legible y mantenible. Y esto hace valer todo el esfuerzo.

Todo el código mencionado en el artículo está disponible mediante este gist.

#Code Snippet #Modelo de datos de objetos #Caché

URL de fuente:https://es.community.intersystems.com/post/objetos-de-proxy-remoto-mediante-despacho-din%C3%A1mico