Artículo
Mathew Lambert · 15 feb, 2021 Lectura de 9 min

Depuración y Gestión robusta de errores en ObjectScript

Introducción y motivación

Una unidad de código en ObjectScript (pongamos, un ClassMethod) puede producir una gran variedad de efectos secundarios inesperados cuando interactúa con partes del sistema que están fuera de su propio alcance y no han sido depuradas adecuadamente. En forma de lista parcial, se incluyen:

- Transacciones
- Locks
- Dispositivos E/S
- Cursores SQL
- Flags de systema y configuración
- $Namespace
- Archivos temporales

Utilizar estas importantes funciones del lenguaje sin hacer cleanup adecuadamente y con desarrollo defensivo podría ocasionar que una aplicación, que normalmente funciona correctamente, falle de forma inesperada y sea difícil de debugar. Es fundamental que el código de cleanup funcione correctamente en todos los posibles casos de error, en especial porque es probable que en pruebas superficiales se ignoren los casos de error. En este artículo se detallan varios problemas conocidos, y se explican dos patrones para lograr que los errores se gestionen y eliminen de manera eficiente.

*Injerto importante que está parcialmente relacionado: ¿quieres asegurarte de que estás probando todos tus casos hasta el límite? Echa un vistazo a la ¡*Herramienta de cobertura para pruebas en Open Exchange*!*

Trampas a evitar

Transacciones

Un enfoque natural y simplista de las transacciones consiste en envolver la transacción en un bloque try/catch, con un TRollback en catch, de la siguiente manera:

```
Try {
    TSTART
    // ... do stuff with data ...
    TCOMMIT
} Catch e {
    TROLLBACK
    // e.AsStatus(), e.Log(), etc.
}
```

Estando aislados, este código siempre que lo escrito entre TStart y TCommit arroje excepciones en vez de producir Quit cuando se produzca un error es perfectamente válido. Sin embargo, es arriesgado por dos razones:

- Si otro desarrollador agrega un “ Quit ” dentro del bloque Try, una transacción se quedará abierta. Sería

sencillo saltarse ese cambio durante la revisión del código, especialmente si no es obvio que haya transacciones involucradas en el contexto actual.
- Si se llama al método con este bloque desde una transacción externa, TRollback restaurará todos los niveles de la transacción.

Un mejor enfoque consiste en rastrear el nivel de la transacción al principio del método y retroceder hasta ese nivel de la transacción al final. Por ejemplo:

```
Set tInitTLevel = $TLevel
Try {
    TSTART
    // ... do stuff with data ...
    // The following is fine now; tStatus does not need to be thrown as an exception.
    If $$$ISERR(tStatus) {
        Quit
    }
    // ... do  more stuff with data ...
    TCOMMIT
} Catch e {
    // e.AsStatus(), e.Log(), etc.
}
While $TLevel > tInitTLevel {
    // Just roll back one transaction level at a time.
    TROLLBACK 1
}
```

Locks

Cualquier código que utilice locks progresivos también debe garantizar que disminuyan los locks en el código de depuración cuando ya no sean necesarios; de lo contrario, dichos locks se mantendrán hasta que el proceso termine. Los locks no deben filtrarse fuera de un método, a menos que la obtención de dicho lock sea un efecto secundario documentado del método.

Dispositivos de E/S

De forma similar, los cambios en el dispositivo de E/S actual (la variable especial $io) tampoco deberían filtrarse fuera de un método, a menos que el propósito del método sea cambiar el dispositivo actual (por ejemplo, habilitar la redirección de E/S). Cuando se trabaja con archivos, es preferible utilizar el paquete %Stream en vez del E/S para archivos secuenciales directos por medio de OPEN / USE / READ / CLOSE. En otros casos, donde es necesario utilizar dispositivos de E/S, debe restablecerse el dispositivo original cuando finalice el método. Por ejemplo, el siguiente patrón de código es arriesgado:

```
Method ReadFromDevice(pSomeOtherDevice As %String)
{
    Open pSomeOtherDevice:10
    Use pSomeOtherDevice
    Read x
    // ... do complicated things with X ...
    Close pSomeOtherDevice
}
```

Si se lanza una excepción antes de que pSomeOtherDevice esté cerrado, entonces $io se quedará como pSomeOtherDevice; esto probablemente ocasionará errores en cascada. Además, cuando el dispositivo se cierra, $io se restablece al dispositivo predeterminado del proceso, el cual probablemente no sea el mismo dispositivo

que se utilizó antes de que se llamara el método.

Cursores SQL

Cuando se utiliza SQL basado en cursores, el cursor debe estar cerrado en caso de ocurra cualquier error. Cuando el cursor no se cierra, pueden producirse filtraciones en los recursos (según la documentación de apoyo). Además, en algunos casos, si se ejecuta el código de nuevo y se intenta abrir el cursor, se obtendrá un error " already open " (SQLCODE -101).

Flags de sistema y configuración

Excepcionalmente, el código de la aplicación puede necesitar que se modifiquen flags a nivel de proceso o del sistema; por ejemplo, muchos están definidos en %SYSTEM.Process y %SYSTEM.SQL . En todos esos casos, debe tenerse cuidado de almacenar el valor inicial y restablecerlo al final del método.

$Namespace

El código que modifica el namespace siempre debe ser New $Namespace al principio, para garantizar que los cambios en el namespace no se filtran fuera del alcance del método.

Archivos temporales

El código de la aplicación encargado de crear archivos temporales, como %Library.File:TempFilename (que, en InterSystems IRIS, realmente crea el archivo), debería eliminar también los archivos temporales cuando ya no se necesiten.

Patrón recomendado: Try-Catch (-Finally)

Muchos lenguajes tienen una función en la que una estructura de tipo try/catch también puede tener un bloque " finally " que se ejecuta cuando se ha completado try/catch, tanto si se produjo una excepción como si no. ObjectScript no lo hace, pero puede aproximarse. Un patrón general para esto, que demuestra muchos de los posibles casos problemáticos, es el siguiente:

```
ClassMethod MyRobustMethod(pFile As %String = "C:\foo\bar.txt") As %Status
{
    Set tSC = $$$OK
    Set tInitialTLevel = $TLevel
    Set tMyGlobalLocked = 0
    Set tDevice = $io
    Set tFileOpen = 0
    Set tCursorOpen = 0
    Set tOldSystemFlagValue = ""

    Try {
        // Lock a global, provided a lock can be obtained within 5 seconds.
        Lock +^MyGlobal(42):5
        If '$Test {
            $$$ThrowStatus($$$ERROR($$$GeneralError,"Couldn't lock ^MyGlobal(42)."))
        }
        Set tMyGlobalLocked = 1

        // Open a file
        Open pFile:"WNS":10
        If '$Test {
```

```
            $$$ThrowStatus($$$ERROR($$$GeneralError,"Couldn't open file "_pFile))
        }
        Set tFileOpen = 1

        // [ cursor MyCursor declared ]
        &;SQL(OPEN MyCursor)
        Set tCursorOpen = 1

        // Set a system flag for this process.
        Set tOldSystemFlagValue = $System.Process.SetZEOF(1)

        // Do the important things...
        Use tFile

        TSTART

        // [ ... lots of important and complicated code that changes data here ... ]

        // All done!

        TCOMMIT
    } Catch e {
        Set tSC = e.AsStatus()
    }

    // Finally {

    // Cleanup: system flag
    If (tOldSystemFlagValue '= "") {
        Do $System.Process.SetZEOF(tOldSystemFlagValue)
    }

    // Cleanup: device
    If tFileOpen {
        Close pFile
        // If pFile is the current device, the CLOSE command switches $io back to the
 process's default device,
        // which might not be the same as the value of $io was when the method was ca
lled.
        // To be extra sure:
        Use tDevice
    }

    // Cleanup: locks
    If tMyGlobalLocked {
        Lock -^MyGlobal(42)
    }

    // Cleanup: transactions
    // Roll back one level at a time up to our starting transaction level.
    While $TLevel > tInitialTLevel {
        TROLLBACK 1
    }

    // } // end "finally"
    Quit tSC
}
```

Nota: en este enfoque, es fundamental que se utilice " Quit " y no " Return " en el bloque " Try " ...; " Return " haría un bypass del cleanup.

Patrón recomendado: Objetos Registrados y Destructores

A veces, el código de depuración puede complicarse. En estos casos, quizás tenga sentido facilitar la reutilización del código de depuración integrándolo en un registered object. El estado del sistema es inicializado cuando se inicializa el objeto o cuando se llama a los métodos del objeto que cambian el estado, y regresa a su valor original cuando el objeto ya no está al alcance. Mira este sencillo ejemplo, que administra las transacciones, el namespace actual y el estado de $System.Process.SetZEOF:

```
/// When an instance of this class goes out of scope, the namespace, transaction leve
l, and value of $System.Process.SetZEOF() that were present when it was created are r
estored.
Class DC.Demo.ScopeManager Extends %RegisteredObject
{

Property InitialNamespace As %String [ InitialExpression = {$Namespace} ];

Property InitialTransactionLevel As %String [ InitialExpression = {$TLevel} ];


Property ZEOFSetting As %Boolean [ InitialExpression = {$System.Process.SetZEOF()} ];


Method SetZEOF(pValue As %Boolean)
{
    Set ..ZEOFSetting = $System.Process.SetZEOF(.pValue)
}

Method %OnClose() As %Status [ Private, ServerOnly = 1 ]
{
    Set tSC = $$$OK

    Try {
        Set $Namespace = ..InitialNamespace
    } Catch e {
        Set tSC = $$$ADDSC(tSC,e.AsStatus())
    }

    Try {
        Do $System.Process.SetZEOF(..ZEOFSetting)
    } Catch e {
        Set tSC = $$$ADDSC(tSC,e.AsStatus())
    }

    Try {
        While $TLevel > ..InitialTransactionLevel {
            TROLLBACK 1
        }
    } Catch e {
        Set tSC = $$$ADDSC(tSC,e.AsStatus())
    }

    Quit tSC
}
```

```
}
```

La siguiente clase demuestra cómo la clase registrada anteriormente podría utilizarse para simplificar la depuración cuando finalice el método:

```
Class DC.Demo.Driver
{

ClassMethod Run()
{
    For tArgument = "good","bad" {
        Do ..LogState(tArgument,"before")
        Do ..DemoRobustMethod(tArgument)
        Do ..LogState(tArgument,"after")
    }
}

ClassMethod LogState(pArgument As %String, pWhen As %String)
{
    Write !,pWhen," calling DemoRobustMethod("_$$$QUOTE(pArgument)_"):"
    Write !,$c(9),"$Namespace=",$Namespace
    Write !,$c(9),"$TLevel=",$TLevel
    Write !,$c(9),"$System.Process.SetZEOF()=",$System.Process.SetZEOF()
}

ClassMethod DemoRobustMethod(pArgument As %String)
{
    Set tScopeManager = ##class(DC.Demo.ScopeManager).%New()

    Set $Namespace = "%SYS"
    TSTART
    Do tScopeManager.SetZEOF(1)
    If (pArgument = "bad") {
        // Normally, this would be a big problem. In this case, because of tScopeMana
ger, it isn't.
        Quit
    }
    TCOMMIT
}

}
```

#Gestión de errores #Mejores prácticas #ObjectScript #Caché #InterSystems IRIS

URL de fuente:https://es.community.intersystems.com/post/depuraci%C3%B3n-y-gesti%C3%B3n-robusta-de-errores-en-objectscript