WCF [DataContractSerializer]- Comment sérialiser une grappe d'objet métier avec des références cycliques ?

Silver Nakache 04 Juillet 2007.

Introduction

Lors du design des classes et des interfaces, il est d' usage de découpler les implémentations des contrats.

En effet, définir un ou plusieurs contrats formalise le protocole qui doit être mis en place pour établir une discussion entre les différentes couches logicielles. Dans une approche véritablement industrielle, l'architecte veille à ce que chaque couche puisse être validée par des implémentations différentes : des « mocks » dans un premier temps, des implémentations réelles bien sûr, des implémentations optimisées par la suite. Il veille aussi à conserver une architecture saine en adressant la problématique de « Levelization » du code.

Lorsque l'architecture est suffisamment bien découpée ; elle dispose d'une couche assez étoffée d'objets métiers, qui ont pour vocation d'être proche du monde réel afin d'adresser les fonctionnalités métiers le plus finement possible.

Il n'est pas rare de trouver des dépendances entre ces objets qui produisent des références cycliques, par exemple une propriété 'parent' d'un objet. Du point de vue de la programmation, il s'agit de permettre la 'navigation' entre les différents objets.

Dans un modèle orienté 'message' comme WCF, qu'advient-il de la sérialisation de tels objets ?

Prenons le contrat de service suivant :

```
namespace ContractServicesInterfaceDefinition
{
    [ServiceContract]
    [ServiceKnownType(typeof(MyObject))]
    public interface IMyContract
    {
        [OperationContract]
        IMyObject GetMyObject();
    }
```

```
 }
```

La méthode 'GetMyObject' renvoie un objet qui contient une dépendance cyclique dans l'interface IMyObject à savoir :

```
namespace DomainInterfaces
{
    public interface IMyObject
    {
        String GetMyString();
        IMyObject GetCircularStuff();
    }
}
```

Voici une implémentation :

```
namespace DomainImplementation
{
        [DataContract]
        public class MyObject : IMyObject
        {
            [DataMember]
            String _MyString;

            [DataMember]
            IMyObject _CircularRef;

            public MyObject()
            {
                _MyString = "Hello world";
                _CircularRef = this;
            }

            public String GetMyString() { return _MyString; }

            public IMyObject GetCircularStuff() { return _CircularRef; }

        }

}
```

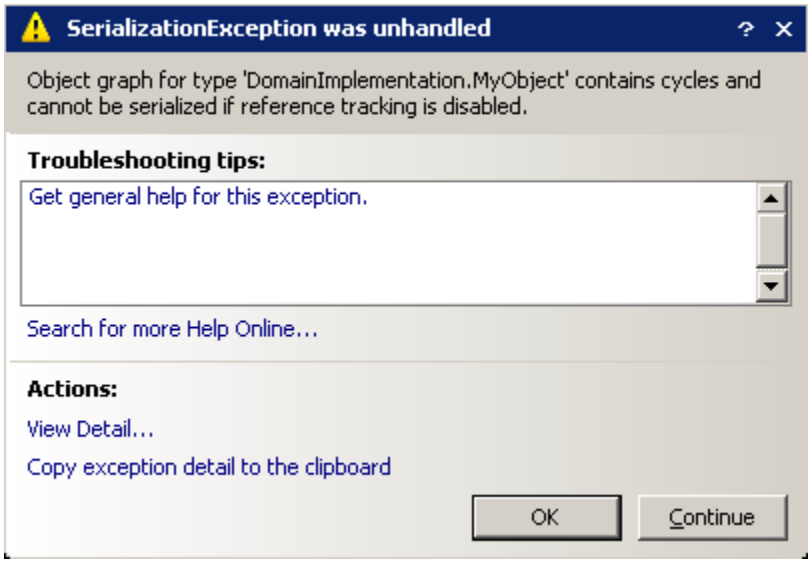
Lors de la ré-hydratation de l'objet, côté client, il convient de restituer un 'pointeur' à l'endroit du retour de la méthode `GetCircularStuff()`. Ainsi, il apparait clairement pour le client que l'objet est utilisé de manière locale ou distante.

Identification de la problématique

Les sous-systèmes de Serialization/Déserialization WCF sont désormais très aboutis. Les leçons sur les types difficiles à serializer du XMLSerializer ou du BinarySerializer sont maintenant très correctement adressés par les nouveaux « DataContractSerializer » et « NetDataContractSerializer ».

« Si nous laissons la sous-couche de serialization opérer, nous obtenons un message d'erreur assez explicite :

« Object graph for type 'DomainImplementation.MyObject' contains cycles and cannot be serialized if reference tracking is disabled. »



Par défaut, c'est le DataContractSerializer qui est selectionné et celui-ci ne préserve pas les références cycliques.

Il existe deux solutions :

1- Soit utiliser le NetDataContractSerializer qui préserve les références par défaut, c'est effectivement ce que fait l'usage de l'attribut `[EmbedTypeInSerializer]` (cf. Article sur l'attribut ce sujet).
2- Soit informer la sous-couche de serialisation d'utiliser un DataContractSerializer qui préserve les références.

Dans ce deuxième cas, il n'y a pas de solution 'B' (Binding), c'est-à-dire pas de possibilité de configuration. Cela doit être résolu en changeant le comportement 'Behavior' au niveau de l'opération, le contrat de service ressemblerait à cela :

```
namespace ContractServicesInterfaceDefinition
{
    [ServiceContract]
    [ServiceKnownType(typeof(MyObject))]
    public interface IMyContract
    {
        [OperationContract]
        [PreserveDataContractReference]
        IMyObject GetMyObject();
    }
}
```

Solution Technique

Observons le résultat d'une sérialisation avec le DataContractSerializer avec le nouveau « Behavior » induit par la présence du nouvel'attribut `[PreserveDataContractReference]`.

```
<MyObject z:Id="1">

       <_CircularRef z:Ref="1" i:nil="true"/>

       <_MyString z:Id="2">Hello world</_MyString>

</MyObject>
```

Les attributs `z:Id="1" et z:Ref="1" de _CircularRef` permettent désormais de reconstituer le 'pointeur' lors de la réhydratation et de restituer la grappe d'objets telle qu'elle est attendue.

Implémentation du nouvel attribut `[PreserveDataContractReference]` :

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Xml.Serialization;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.Runtime.Serialization;
using System.ServiceModel.Description;


namespace WCFCustomAttributes
{
    [AttributeUsage(AttributeTargets.Method)]
    public class PreserveDataContractReferenceAttribute : Attribute, IOperationBehavior
    {

#region IOperationBehavior Members

        public void AddBindingParameters(OperationDescription description,
BindingParameterCollection parameters)
        {

        }

        public void ApplyClientBehavior(OperationDescription description,
System.ServiceModel.Dispatcher.ClientOperation proxy)
        {
            PreserveReferenceDataContractSerializerOperationBehavior(description);
        }

        public void ApplyDispatchBehavior(OperationDescription description,
System.ServiceModel.Dispatcher.DispatchOperation dispatch)
        {
            PreserveReferenceDataContractSerializerOperationBehavior(description);
        }

        public void Validate(OperationDescription description)
        {

        }

#endregion


        private static void
PreserveReferenceDataContractSerializerOperationBehavior(OperationDescription description)
        {
            DataContractSerializerOperationBehavior dcsOperationBehavior =
description.Behaviors.Find<DataContractSerializerOperationBehavior>();
            if (dcsOperationBehavior != null)
            {
                description.Behaviors.Remove(dcsOperationBehavior);
```

```
                description.Behaviors.Add(new
PreserveDataContractReferenceOperationBehavior(description));
            }
        }

        class PreserveDataContractReferenceOperationBehavior :
DataContractSerializerOperationBehavior
        {
            public PreserveDataContractReferenceOperationBehavior(OperationDescription
operationDescription) : base(operationDescription) { }
            public override XmlObjectSerializer CreateSerializer(Type type, string name, string
ns, IList<Type> knownTypes)
            {

                return new DataContractSerializer(type, name, ns, knownTypes, 0x7FFF,
false/*ignoreExtensionDataObject*/, true/*preserveObjectReferences*/,
null/*dataContractSurrogate*/);
            }

            public override XmlObjectSerializer CreateSerializer(Type type, XmlDictionaryString
name, XmlDictionaryString ns, IList<Type> knownTypes)
            {
                return new DataContractSerializer(type, name, ns, knownTypes, 0x7FFF,
false/*ignoreExtensionDataObject*/, true/*preserveObjectReferences*/,
null/*dataContractSurrogate*/);
            }
        }
    }
}
```

En conclusion

Dans bien des cas, il est préférable d'utiliser le NetDataContractSerializer, il montre ici sa supériorité en termes de sérialisation de grappe d'objets. L'inconvénient majeur est qu'il est plus verbeux au niveau des messages transmis. Le DataContractSerializer peut-être enrichi pour permettre le référencement entre objets.

Dans les prochaines versions, il serait judicieux de pouvoir agir sur le 'B' de l'ABC de WCF pour permettre de le configurer la propriété '`preserveObjectReferences`' par fichier.