

INTRODUCTION A LA SERIALISATION

Cet article est composé de ce document et d'un fichier contenant les sources des démonstrations vous permettant d'approfondir quelque peu vos connaissances.

A noter qu'il est essentiel de tester et de lire les commentaires de code des démonstrations pour bien comprendre cet article.

1. Les concepts

La sérialisation

C'est un processus de sauvegarde de l'état d'un objet à destination d'une zone de stockage.

La dé-sérialisation

Il s'agit d'un processus de conversion d'un flux d'octets stockés en un objet actif.

Auparavant, les développeurs devaient gérer ce mécanisme manuellement. Avec l'arrivée de .Net, leur tâche s'en trouve simplifiée. Le Framework .NET fournit un riche support pour ces opérations, comme nous pourrons le constater à travers les démonstrations.

2. Des exemples d'utilisations

Utilisé par exemple dans les applications windows clientes

La sérialisation peut être employée pour sauvegarder l'état d'une application. On peut faire appel à la sérialisation/dé-sérialisation lors d'opérations de copier/coller des données (objet) dans le presse-papier. On peut aussi l'utiliser pour cloner totalement un objet complexe.

Le remoting .net


La sérialisation/dé-sérialisation est un processus essentiel dans le remoting. Le but est de partager des objets (notion d'état) à travers le réseau. Le fait que la plateforme offre facilement un support de la sérialisation/dé-sérialisation simplifie grandement la mise en place rapide d'objets distribués.

Les Web Services

Dans le cadre des Web Services, une classe est particulièrement adaptée au mécanisme de sérialisation : Il s'agit de XMLSerializer. Grâce aux différents attributs, on peut facilement contrôler le flux Xml.

3. La sérialisation et la persistance

Il ne faut pas confondre ces termes :

-  La sérialisation est le mécanisme de transformation d'un objet ou d'un groupe d'objets en un flux d'octet. Il n'y a pas ici de notion de stockage.

.NET passionnément, tout simplement

🚦 La persistance fait référence au stockage permanent de données.

4. Prise en charge des types de base

Les types de base de la plateforme .Net supportent la sérialisation. Des objets plus évolués comme le DataSet, ou comme l'image (pas un grand intérêt pour la notion d'état) supportent aussi la sérialisation.

5. Quelle est le mécanisme ? Comment cela fonctionne t-il ?

🚦 Les formateurs

Les formateurs jouent le rôle suivant : Ils prennent en charge un objet (sérialisable) et le sérialise en un flux.

Il existe deux formateurs fournis en standard sur la plateforme :

🚦 le BinaryFormatter

Il y a émission d'octets en flux continu à partir d'un ou plusieurs objets.

🚦 Le SoapFormatter

Il y a émission d'un flux xml (donc parfaitement compréhensible) en utilisant les spécifications SOAP. Le XML étant très verbeux, on peut s'attendre à une vitesse, des performances moindres, ce qui est bien le cas. Pour utiliser ce formateur, il est nécessaire d'ajouter une référence à l'assembly Serialisation.Formatters.Soap.dll

🚦 Peut-on créer son propre formateur ?

Pour sérialiser avec son propre formateur, il faut créer une classe implémentant l'interface IFormatter. On peut aussi dériver de la classe abstraite Formatter.

Il n'est pas si facile de réaliser son propre formateur, surtout dès qu'on s'attaque à des objets complexes (Pour ma part, j'ai connu quelques problèmes avec le constructeur de la classe sérialisable).

Les formateurs de base sont très souples puisqu'on peut les utiliser pour manipuler des références (référence utilisé par plusieurs objets), des membres de classes de type public ou private, de multiples références vers un seul objet (exemple en .net remoting du mode Server Activated in singleton).

6. DEMO 1 : Sérialisation et dé-sérialisation d'un objet simple

La Démonstration 1 met en évidence la simplicité d'utilisation des outils de sérialisation/dé-sérialisation. L'objet à sérialiser doit comporter l'attribut [Serializable()]. Tout objet utilisé dans la classe doit être sérialisable.

On notera que la sérialisation binaire est bien évidemment plus rapide que la sérialisation SOAP.

.NET passionnément, tout simplement

```
//Désérialisation en binaire

//Objet à sérialiser : _cdt

//Sérialisation binaire
BinaryFormatter fmt = new BinaryFormatter();
MemoryStream s = new MemoryStream();
BinaryReader br = new BinaryReader(s);

//sérialisation : 3 à 4 fois moins de temps que pour Soap (Test avec DevPartner Profiler)
fmt.Serialize(s,_cdt);

s.Position = 0; //important de repositionner !
//Désérialisation
Candidat c = (Candidat)fmt.Deserialize(s);//la désérialisation est coûteuse, le cast aussi

textBoxSortie.Text = c.ToString();

//fermeture du flux et du Reader
br.Close();
s.Close();
```

7. DEMO 2 : Sérialisation et désérialisation d'un objet plus complexe avec références croisées.

La Démonstration 2 montre l'efficacité des outils de sérialisation/dé-sérialisation sur des objets plus complexes, en particulier une hiérarchie d'objets.

Nous avons pris l'exemple de candidats se présentant à des examens. Au final, nous avons un objet examens contenant les différentes épreuves avec la liste des candidats, des candidats pouvant se présenter à plusieurs épreuves.

On regardera en particulier le mécanisme de référence dans le flux Soap après la sérialisation Soap.

On notera que l'on peut indiquer à certains membres d'une classe de ne pas être sérialisé (voir class Candidat et l'image) via l'attribut [NonSerialized]. On pourra s'amuser à désactiver cet attribut pour voir le flux généré (l'Image fait partie des objets sérialisables de la plate-forme).

On constatera que la sérialisation d'une classe sérialise tous les éléments de la classe y compris les champs privés ! Quand l'objet est dé-sérialisé, il retrouve parfaitement son état initial. Dans un souci de coût d'opération, on veillera à ne conserver que l'état nécessaire de certains éléments de la classe. On abusera donc de l'attribut <NonSerialized> pour un souci de performance.

Rappelons une évidence : Lorsqu'on dé-sérialise, il faut utiliser le même Formateur que pour la sérialisation. Sinon, une exception est levée.

.NET passionnément, tout simplement

On résout ce problème grâce à la sérialisation/désérialisation. A cet effet, on va utiliser le formateur le plus efficace : le BinaryFormatter.

9. DEMO 3 : Illustration du clonage partiel et du clonage total

En résumé, si on veut fournir la possibilité de cloner un objet, il suffit d'implémenter l'interface ICloneable.

Si l'objet est complexe, cela ne fournira qu'une copie partielle de l'objet. Dans ce cas, il est nécessaire de sérialiser/désérialiser l'objet pour obtenir un véritable clone de l'objet (encore appelé « deep clone »). La démonstration vous donne un exemple des deux types de clonage.

```
//clonage partiel (shallow copy)
public object Clone()
{
    return this.MemberwiseClone();
}

//clonage total (deep clone)
public object PerfectClone()
{
    MemoryStream ms = new MemoryStream();
    object objResult;

    BinaryFormatter bf = new BinaryFormatter();
    bf.Serialize(ms, this);

    ms.Position = 0;
    objResult = bf.Deserialize(ms);
    ms.Close();
    return objResult;
}
```

10. Personnalisation du processus de sérialisation

Grâce à l'attribut [Serializable()], on accède à un nombre de fonctionnalité facilement. Toutefois, si l'on souhaite ne pas utiliser le comportement par défaut, on doit implémenter alors l'interface ISerializable et créer de toute pièce le processus de sérialisation. Ce n'est pas quelque chose d'insurmontable comme on va le voir.

On peut avoir besoin d'implémenter son propre formatage pour réduire les coûts de stockage ou encore sécuriser certains objets.

Pour cela, on doit implémenter la méthode GetObjectData pour la sérialisation. Et on doit aussi avoir un constructeur avec des paramètres particuliers pour permettre la dé-sérialisation. Par les deux méthodes passent un objet de type SerializationInfo.

.NET passionnément, tout simplement

L'objet `SerializationInfo` agit comme une sorte de conteneur pour les données sérialisés. L'objet fournit des méthodes qui ajoutent ou rétablissent des données sérialisées.

Généralement, il n'est pas nécessaire de faire sa propre sérialisation. La sérialisation fournie par le Framework suffit dans la plupart des cas.

11. DEMO 4 : Cryptage de certaines données lors d'une Sérialisation Soap

Cette démonstration montre que l'on peut personnaliser le processus de sérialisation. Ici, nous appliquons un cryptage sur le password d'un objet utilisateur comportant un login et un mot de passe.

On s'aperçoit que la personnalisation se fait avec une étonnante facilité.

```
[Serializable()]
public class User : ISerializable
{
    private string _password;
    private string _login = "";
    public User(string login, string password)
    {
        _login = login;
        _password = password;
    }

    //ne pas oublier le constructeur privé !
    private User(SerializationInfo info, StreamingContext context)
    {
        _login = info.GetString("Login");
        _password = Crypteur(info.GetString("Password"), false);
    }

    [SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("Login", _login);
        info.AddValue("Password", Crypteur(_password, true));
    }
}
```

12. DEMO 5 : Le cas du Hashtable sur le Framework 1.1

Cette démonstration pour vous montrer une curiosité (je n'ose pas dire un bug). Imaginez que vous avez créé un objet complexe avec notamment un Hashtable dans cet objet. Et bien, vous pourrez sérialiser la totalité de l'objet sans problème, par contre, vous ne dé-sérialiserez pas la totalité de l'objet.

J'ai passé plusieurs heures avant de comprendre que le Hashtable bien qu'inplémentant l'interface `ISerializable`, se sérialise très bien, par contre, la désérialisation ne se fait pas du tout. Cette remarque vous évitera sans doute quelques heures de recherches... Gageons que cette particularité de la version 1.1 du Framework, sera corrigée dans la version 2.0

.NET passionnément, tout simplement

```
[Serializable()]
public class Users : ISerializable
{
    private Hashtable _hash;
    private string _nomGroupe = "";
    public Users(string nomGroupe)
    {
        _hash = new Hashtable();
        _nomGroupe = nomGroupe;
    }

    //ne pas oublier le constructeur privé !
    private Users(SerializationInfo info, StreamingContext context)
    {
        if (_hash == null)
        {
            _hash = new Hashtable();
        }

        _nomGroupe = info.GetString("Nom");
        //la dé-sérialisation du HashTable ne fonctionne pas ;-(
        Hashtable h = (Hashtable) info.GetValue("hash", _hash.GetType());

        if (h.Count!=0)
        {
            IEnumerator ie = h.Keys.GetEnumerator();

            while (ie.MoveNext())
            {
                this.Add((string) ie.Current, Crypteur((string) h[ie.Current], false));
            }
        }
    }
}
```

13. DEMO 6 : De l'usage de XmlSerializer

Cette démonstration a pour but de vous montrer la simplicité d'utilisation de la classe XmlSerializer. On prêtera l'attention sur l'attribut XmlIgnore qui permet de ne pas rendre persistant certaines données.

A noter qu'il existe toute une gamme d'attributs qui permettent de personnaliser le flux Xml lors de la sérialisation.

.NET passionnément, tout simplement



14. En savoir plus :

XmlSerializer :

<http://alain.vizzini.free.fr/article01.html>

XML Serialization in the Framework .NET

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexxml/html/xml01202003.asp>

.NET Remoting :

<http://www.dotnet-tech.com/tutoriels/remoting/>

Contacter l'auteur :

Frédéric Mélantois

Email : fmelantois@free.fr