Présentation de la classe SortedSet du .NET Framework 4

par Jérôme Lambert (Espace perso) (Blog)

Date de publication : 25 juin 2010

Dernière mise à jour :

Cet article vous permettra de découvrir la nouvelle classe SortedSet<T> introduite avec le .NET Framework 4 de Microsoft et qui permet de gérer nativement une séquence triée d'objets.

N'hésitez pas à laisser votre avis sur le contenu de l'article directement via notre forum :

I - Introduction

Chaque nouvelle version du Framework .NET de Microsoft apporte son lot de nouveautés et de possibilités avec l'ajout de nouvelles classes qui pour certaines intégreront de nouvelles fonctionnalités et pour d'autres faciliteront tout simplement nos développement. Le Framework .NET 4 ne deroge pas à la regle. Dans cet article, nous allons découvrir les possibilités d'une toute nouvelle classe introduite dans la famille des collections : SortedSet<T>. Soyons clair dès le départ, cette classe n'est pas une révolution mais au niveau de la facilité de nos développement, elle va pouvoir nous donner un bon petit coup de main ! Pour faire bref, cette classe permet de trier nativement une séquence d'objets unique et sans altérer pour autant les performances - du moins, c'est que qu'affirme Microsoft dans sa documentation, nous verrons plus tard si cela s'avère vrai.

II - Mise en situation

Prenons un exemple simple : "*soit une liste de chaînes de caractères, comment trier cette liste ?*"

Avec la classe List<T>, cela ne s'avère pas trop compliqué car elle propose une méthode "*Sort*" qui permet de réorganiser les éléments de la liste avec ordonancement croissant.

Ainsi, l'exemple suivant :

```
List<string> names = new List<string>();
names.Add("Jerome");
names.Add("Philippe");
names.Add("Benjamin");
names.Add("Louis-Guillaume");
names.Add("Thomas");
names.Add("Nicolas");
names.Add("Vincent");
names.Add("Florian");

names.Sort();

foreach (string name in names)
{
    Console.WriteLine(name);
}

Console.Read();
```

Donnera le résultat suivant :

```
Benjamin
Florian
Jerome
Louis-Guillaume
Nicolas
Philippe
Thomas
Vincent
```

Jusque là parfait mais petite question quand même : "*ai-je parlé qu'il fallait faire d'un tri croissant ? Comment faire pour obtenir un tri décroissant par exemple ?*"

Là encore, le Framework .NET propose depuis le début une solution à ce problème grâce à l'interface "*IComparer*". Cette interface expose une méthode "*Comparer*" qu'il suffit d'implémenter pour définir sa propre comparaison entre deux objets.

Implémentons cette interface dans une nouvelle classe (vous remarquez au passage l'utilisation de la version générique de cette interface disponible depuis le Framework .NET 2.0) :

```
public class MonComparateurDeNomDecroissant : IComparer<string>
{

    public int Compare(string x, string y)
    {
        // on inverse les deux objets pour la comparaison
        return y.CompareTo(x);
    }
}
```

Il ne reste plus qu'à utiliser notre méthode "*Sort*" de tout à l'heure qui offre une autre signature acceptant comme argument un "*IComparer*".

Notre exemple devient donc :

```
List<string> names = new List<string>();
names.Add("Jerome");
names.Add("Philippe");
names.Add("Benjamin");
names.Add("Louis-Guillaume");
names.Add("Thomas");
names.Add("Nicolas");
names.Add("Vincent");
names.Add("Florian");

names.Sort(new MonComparateurDeNomDecroissant());

foreach (string name in names)
{
    Console.WriteLine(name);
}

Console.Read();
```

Avec comme résultat :

```
Vincent
Thomas
Philippe
Nicolas
Louis-Guillaume
Jerome
Florian
Benjamin
```

Notre liste de noms est à présent triée par ordre alphabétique décroissant.

Si on désire utiliser trier des listes d'objets personalisés (donc différents des types primitifs comme int, float, string, etc.), c'est le même principe avec l'utilisation de l'interface "IComparer".

A présent, question subsidiaire : "*Comment faites-vous pour réoarganiser votre liste liste dès l'ajout d'un élément ? Voir encore plus compliqué en modifiant tout simplement un des éléments de votre liste ?*"

On pourrait créer son propre type collection qui encapsulerait une type collection du Framework .NET par exemple ou qui hériterait des interfaces nécessaires. Pour prendre l'exemple de la classe List<T> du Framework .NET, elle implémente les interfaces suivantes : *IList<T>*, *ICollection<T>*, *IEnumerable<T>*, *IList*, *ICollection*, *IEnumerable*. Comme vous pouvez le constater, ce n'est pas évident comme développement et pourtant la fonctionnalité nécessaire est assez basique.

C'est là qu'intervient la nouvelle classe "*SortedSet<T>*" !

III - La classe SortedSet

La classe "*SortedSet*" fait son apparition dans l'espace de noms "System.Collections.Generic", nous avons donc affaire à une nouvelle classe générique. Comme expliqué en introduction de cet article, la classe "*SortedSet<T>*" permet de maintenir une liste triée d'objets uniques sans dégrader les performances peu importe que les opérations soient une insertion, une suppression ou encore une lecture. Cependant, les possibilités de cette classe ne s'arrêtent pas là car il est aussi possible de manipuler un sous ensemble de ces éléments triés ce que nous verrons assez vite.

Nous allons commencer par créer notre première instance de cette nouvelle classe en se basant sur notre exemple de mise en situation concernant une liste de noms.

```
SortedSet<string> namesSet = new SortedSet<string>()
                                    {
                                        "Jerome",
                                        "Philippe",
                                        "Benjamin",
                                        "Louis-Guillaume",
                                        "Thomas",
                                        "Nicolas",
                                        "Vincent",
                                        "Florian"
                                    };

foreach (string name in namesSet)
{
    Console.WriteLine(name);
}

Console.Read();
```

Résultat :

```
Benjamin
Florian
Jerome
Louis-Guillaume
Nicolas
Philippe
Thomas
Vincent
```

Comme vous pouvez le constater, les éléments ont été triés dès leur ajout dans notre liste.

III-A - Ajout et Suppression

Allons un petit peu plus loin en ajoutant à notre liste un nouveau nom "Didier" et en supprimant un autre "Vincent".

```
namesSet.Add("Didier");
namesSet.Remove("Vincent");

foreach (string name in namesSet)
{
    Console.WriteLine(name);
}
```

Résultat :

```
Benjamin
Didier
Florian
Jerome
Louis-Guillaume
```

```
Nicolas
Philippe
Thomas
```

Dès l'ajout d'un nouvel élément et la suppression d'un existant, notre liste a été autoamtiquement mise à jour sans avoir eu besoin de faire quoique ce soit.

Une autre méthode pour supprimer des éléments de manière conditonnelle est aussi disponible au niveau de cette classe : "*RemoveWhere*". Elle prend en paramètre une expression lambda.

```
int removeCount = namesSet.RemoveWhere(name => name.Contains("a"));

Console.WriteLine("{0} éléments ont été supprimés", removeCount);
foreach (string name in namesSet)
{
    Console.WriteLine(name);
}
```

Résultat :

```
5 éléments ont été supprimés
Jerome
Philippe
Vincent
```

Dans notre exemple, nous avons une expression lambda qui force la suppression de tous les élements qui contiennent la lettre 'a'.

III-B - Min et Max

III-C - Opérations d'ensembles

III-D - Fusion et Copie

V - Conclusion

Au travers de cet article, vous avez pu découvrir toutes les possibilités de cette nouvelle classe "*Sorted<T>*" et son utilisation.

Comme vous avez pu le constater par vous même, cette classe n'a rien de révolutionnaire en soit mais il est certain que les facilités apportées feront que cette classe fera partie de nos développements de demain.

VI - Remerciements

Je tiens à remercier toute l'équipe .NET pour son aide précieuse pour l'amélioration de cet article.