

## Technique

# Reflection en .net

La " Reflection " est un mécanisme qui permet d'obtenir de nombreuses informations sur une assembly .net, Il faut la considérer comme une API permettant l'interrogation d'une assembly, cette API se trouve dans le namespace System.Reflection de l'assembly mscorlib.

### La classe System.Type

Le type System.Type est la base de l'utilisation de la Reflection, ce type particulier permet de décrire un type. Il existe deux solutions pour récupérer une instance de System.Type : on peut utiliser la méthode GetType qui est accessible à tout objet, car définie dans la classe Object ou alors utiliser le mot clé typeof.

```
Type t = typeof(Person);
```

```
// ou
```

```
Person p = // Récupération de l'objet Person.
```

```
Type t = p.GetType();
```

Si on connaît le nom du type à récupérer lors de la conception du programme, il est conseillé d'utiliser le mot clé typeof. La différence entre les deux est que GetType est une méthode, elle sera utilisée lors de l'exécution alors que typeof est un mot clé du langage et sera donc utilisé lors de la compilation. La solution typeof est donc beaucoup plus performante. Définissons une classe Person qui nous servira tout au long de nos exemples.

```
[Description("Personne")]
```

```
public class Person
```

```
[Description("Age")]
```

```
public int Age { get; set; }
```

```
[Description("Nom")]
```

```
public String LastName { get; set; }
```

```
[Description("Prénom")]
```

```
public String FirstName { get; set; }
```

```
public String ToString(String format)
```

```
return String.Format(format, LastName, FirstName, Age);
```

Pour récupérer toutes les propriétés, du type Person nous allons utiliser la méthode GetProperties de la classe System.Type. Cette méthode nous renvoie un tableau de descripteurs de propriétés : PropertyInfo[]. Parmi les propriétés intéressantes de la classe PropertyInfo il y a les propriétés Name et PropertyType qui nous permettent de connaître le nom de la propriété et le Type de celle-ci.

```
static void Main(string[] args)
```

```
Type t = typeof(Person);  
foreach (PropertyInfo pi in t.GetProperties())
```

```
Console.WriteLine(pi.PropertyType.Name);
```

```
Console.WriteLine("\t");
```

```
Console.WriteLine(pi.Name);
```

```
Console.WriteLine("\n");
```



Il est également possible de lister les différentes méthodes d'un type en utilisant la méthode GetMethods qui nous retourne un tableau de MethodInfo. Comme pour le PropertyInfo, la propriété Name nous retourne le nom de la méthode, la propriété ReturnType nous renvoie le Type de la méthode.

```
static void Main(string[] args)
```

```
Type t = typeof(Person);  
foreach (MethodInfo mi in t.GetMethods())
```

```
Console.WriteLine(mi.ReturnType.Name);
```

```
Console.WriteLine("\t");
```

```
Console.WriteLine(mi.Name);
```

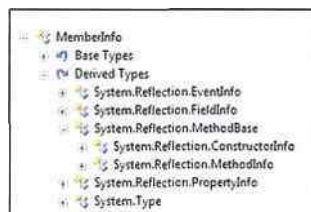
```
Console.WriteLine("\n");
```



On remarque que l'exécution du code ci-dessus, nous retourne la liste des méthodes mais aussi la liste des getter et setter des propriétés. En fait, en interne les propriétés .Net ne sont que des méthodes get\_xxx et set\_xxx, ces méthodes sont cependant inaccessibles au développeur. La méthode GetMethods possède deux signatures : GetMethods() et GetMethods(BindingFlags). La deuxième signature nous permet de spécifier un BindingFlags.

Un *BindingFlags* est une énumération qui permet de spécifier le lieu de recherche, cette énumération est décorée de l'attribut *Flags*, cela signifie qu'il est possible d'effectuer des opérations binaires sur ses valeurs, par exemple combiner plusieurs valeurs grâce au " ou " logique. Concrètement la valeur *BindingFlags.Public | BindingFlags.Instance* indique que l'on doit rechercher seulement les méthodes publiques d'instance. Examinons la première signature de la méthode *GetMethods* dans *Reflector*. *Reflector* est un utilitaire qui permet de décompiler les assemblages .net, il nous permet d'analyser le code du Framework. Nous voyons que cette méthode ne retourne par défaut que les méthodes *public*, *static* et d'instance.

A partir de l'exécution du code ci-dessus, on remarque que deux nouvelles méthodes apparaissent : *MemberwiseClone* et *Finalize*. C'est grâce au *BindingFlags.NonPublic* que ces deux méthodes privées sont visibles. Ci-dessus nous avons vu les classes *PropertyInfo* et *MethodInfo*. Ces deux classes ont un point commun : elles dérivent de *MemberInfo*. En effet, *MemberInfo* est la classe de base permettant de récupérer les informations sur les méthodes, les attributs, les propriétés.



On peut récupérer les événements, les champs et les constructeurs de la même façon que le code plus haut.

### Les attributs

Les propriétés de notre classe *Person* sont décorées de l'attribut *Description*. La *Reflection* nous permet de manipuler les attributs. Pour cela nous allons utiliser la méthode *GetCustomAttributes*, cette méthode est définie au niveau de la classe *MemberInfo*, il est donc possible de récupérer les attributs des types, méthodes, propriétés, etc. avec la même syntaxe. Le premier argument de cette méthode est le type d'attribut à rechercher, le second indique si l'on doit chercher ces attributs dans l'héritage.

```
public static DescriptionAttribute GetDescriptionAttribute(MemberInfo mi)
{
    var attributes = mi.GetCustomAttributes(typeof(DescriptionAttribute), true);
    if (attributes.Length > 0)
        return attributes[0] as DescriptionAttribute;
    else
        return null;
}

static void Main(string[] args)
{
    Type t = typeof(Person);
    Console.WriteLine(t.FullName);
    Console.WriteLine('\t');
    Console.WriteLine(GetDescriptionAttribute(t).Description);
    foreach (PropertyInfo pi in t.GetProperties())
    {
        Console.WriteLine('\t');
        Console.WriteLine(pi.PropertyType.Name);
        Console.WriteLine('\t');
        Console.WriteLine(pi.Name);
        Console.WriteLine('\t');

        Console.WriteLine(GetDescriptionAttribute(pi).Description);

        Console.WriteLine('\n');
    }
}
```

```
Console.WriteLine('\t');
Console.WriteLine(pi.PropertyType.Name);
Console.WriteLine('\t');
Console.WriteLine(pi.Name);
Console.WriteLine('\t');

Console.WriteLine(GetDescriptionAttribute(pi).Description);

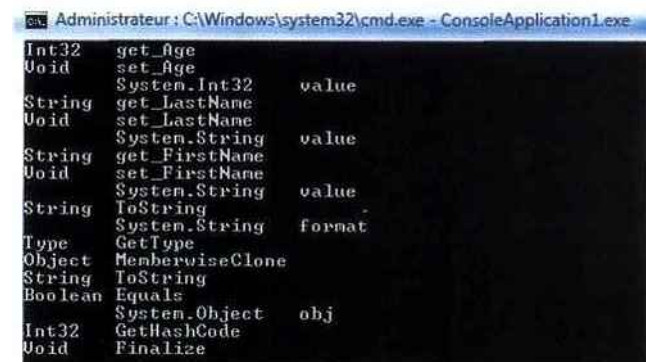
Console.WriteLine('\n');
```



```
Disassembler
public MethodInfo[] GetMethods()
{
    return this.GetMethods(BindingFlags.Public | BindingFlags.Static | BindingFlags.Instance);
}
```

Modifions notre code afin de récupérer les méthodes privées. Pour cela nous utilisons la valeur *BindingFlags.NonPublic* combinée aux valeurs par défaut. Il est également possible de récupérer les paramètres des méthodes en utilisant la méthode *GetParameters* de la classe *MethodInfo*. Cette méthode renvoie un tableau de *ParameterInfo*, grâce à sa propriété *ParameterType* nous obtenons le *Type* du paramètre.

```
static void Main(string[] args)
{
    Type t = typeof(Person);
    foreach (MethodInfo mi in t.GetMethods(BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public | BindingFlags.NonPublic))
    {
        Console.WriteLine(mi.ReturnType.Name);
        Console.WriteLine('\t');
        Console.WriteLine(mi.Name);
        Console.WriteLine('\n');
        foreach (ParameterInfo pi in mi.GetParameters())
        {
            Console.WriteLine('\t');
            Console.WriteLine(pi.ParameterType);
            Console.WriteLine('\t');
            Console.WriteLine(pi.Name);
            Console.WriteLine('\n');
        }
    }
}
```



## Invocation de code

Nous avons vu que nous pouvons récupérer les méthodes d'une classe grâce à la méthode `GetMethods`, si nous recherchons une méthode précise, nous pouvons utiliser la méthode `GetMethod(String [,BindingFlags])`, attention, si la méthode possède plusieurs signature alors il faudra utiliser l'autre signature de `GetMethod` : `GetMethod(String, Type[] [,BindingFlags])` où le tableau de type correspond aux types des arguments de la méthode à rechercher. On peut également récupérer la méthode courante grâce à la méthode static `MethodInfo.GetCurrentMethod()`. On peut aller encore plus loin en récupérant toutes les méthodes de la pile d'appel grâce à la classe `StackTrace`.

```
static void DisplayStackTrace(Boolean test, String toto)
```

```
    StackTrace trace = new StackTrace();  
    foreach (StackFrame frame in trace.GetFrames())  
  
        MethodBase mb = frame.GetMethod();  
  
        Console.WriteLine(mb.Name);  
        foreach (ParameterInfo pi in mb.GetParameters())  
  
            Console.WriteLine('\t');  
            Console.WriteLine(pi.ParameterType.Name);  
            Console.WriteLine('\t');  
            Console.WriteLine(pi.Name);  
  
        Console.WriteLine("\n");
```



A partir d'une instance de `MethodInfo` nous pouvons invoquer la méthode sous-jacente grâce à la méthode `Invoke`. Cette méthode prend en paramètre l'objet sur lequel on veut appeler notre méthode ou `null` si on veut appeler une méthode static puis un tableau de paramètres de la méthode à appeler ou `null` s'il n'y a aucun paramètre.

```
static void Main(string[] args)  
  
    Person p = new Person() { Age = 21, FirstName = "Cyril", LastName = "Durand" };  
  
    Type t = typeof(Person);  
    MethodInfo mi = t.GetMethod("ToString", new Type[] { typeof(String) });  
  
    String result = (String)mi.Invoke(p, new object[] { "{0} {1}" });  
    Console.WriteLine(result);
```

L'enum `BindingFlags` nous permet de récupérer des méthodes `private`, il devient alors possible d'appeler des méthodes `private` !

Il est également possible d'invoquer le getter ou le setter d'une propriété

grâce à son `MethodInfo`, il est cependant plus simple de passer par les méthodes `GetValue / SetValue` du type `PropertyInfo`.

```
static void Main(string[] args)  
{  
    Person p = new Person() { Age = 25, FirstName = "Clément", LastName = "Séry" };  
  
    Type t = typeof(Person);  
    PropertyInfo pi = t.GetProperty("FirstName");  
  
    pi.SetValue(p, "Clément", null);
```

Pour créer une instance d'un objet, il est possible de récupérer le Constructeur via la méthode `GetConstructor` puis d'appeler sa méthode `Invoke`.

```
static void Main(string[] args)  
  
    Type t = typeof(Person);  
  
    ConstructorInfo ci = t.GetConstructor(null);  
    Person p = (Person)ci.Invoke(null);
```

La méthode static `CreateInstance` de la classe `Activator` rend plus élégante la création d'instance, de plus, elle a l'avantage d'être générique :

```
static void Main(string[] args)  
  
    Person p = Activator.CreateInstance<Person>();
```

## Conclusion

L'outil qui reflète le mieux l'utilisation de la " réflexion ", est `Reflector`. Pour ceux qui ne le connaissent pas, `Reflector` est un outil gratuit qui permet de décompiler une assembly .net, il nous permet de parcourir toutes les classes, les méthodes ainsi que le contenu des méthodes d'une assembly. Nous vous invitons vivement à le découvrir, une fois que vous l'aurez essayé vous ne le quitterez plus. L'autre utilisation courante de la Reflection est la création d'un système de plug-in. En effet, dans ce cas, il est nécessaire de charger et d'exécuter dynamiquement des assemblies, cependant le namespace `System.AddIn` de .net 3.5 simplifie grandement le code nécessaire. Dans cet article, nous avons vu les concepts de base de la Reflection : comment obtenir la description d'une assembly.

La Reflection ne s'arrête pas là, grâce au namespace `System.Reflection.Emit`, il nous est possible de créer entièrement une Assembly dynamiquement, en définissant via le code, les types, les méthodes et même le contenu de ces méthodes en manipulant directement le MSIL. Mais attention, la Reflection est un mécanisme coûteux en processeur et en mémoire, à cause de cette lourdeur, il est recommandé de l'utiliser seulement lorsque vous en avez vraiment besoin.



■ Cyril DURAND  
Consultant indépendant,  
CodeS-SourceS

<http://blogs.developpeur.org/cyrl/>

■ Clément SERY  
Ingénieur  
développement  
ITELIOS



<http://www.itelios.com/>