

Optimisation des invocations dynamiques de méthodes en C#

par Romain Verdier (<http://codingly.com>)

Date de publication : 09 juin 2008

Dernière mise à jour :

Cet article a pour but de proposer quelques techniques d'optimisations qui peuvent s'avérer utiles dans la plupart des scénarios impliquant l'invocation dynamiques de méthodes en C#.

1 - Avertissement.....	3
2 - Delegates, caching et memoization.....	3
2.1 - Introduction.....	3
2.2 - Invocations dynamiques de méthodes.....	3
2.3 - Il nous faut la réflexion.....	4
2.4 - Peut-on optimiser ?.....	6
2.5 - Peut-on aller plus loin ?.....	7
2.6 - Conclusion.....	10
3 - Lightweight Code Generation (LCG).....	11
3.1 - Introduction.....	11
3.2 - Mais c'est quoi ce truc ?.....	11
3.3 - Parfois, c'est inutile.....	11
3.4 - Un exemple moins ingrat.....	12
3.5 - Conclusion.....	15

1 - Avertissement

A l'origine, cet article a été rédigé en deux parties pour mon **blog**. J'ai simplement procédé à une fusion sommaire des deux posts avant de le publier ici, ce qui peut expliquer le résultat particulier. J'espère qu'une fois de plus, vous parviendrez à en faire abstraction.

La première partie "Delegates, caching et memoization" (originellement posté **ici**) introduit le sujet des invocations dynamiques et propose plusieurs pistes d'optimisation autour d'un thème commun : le mise en cache.

La seconde partie "Lightweight Code Generation" (originellement posté **ici**) met en avant un exemple dans lequel la première technique d'optimisation est impossible, et décrit une solution basée sur la génération dynamique de code intermédiaire, via l'API System.Reflection.Emit et les DynamicMethod.

2 - Delegates, caching et memoization

2.1 - Introduction

Je travaille actuellement en tant que consultant .NET sur un projet d'une certaine taille.

Travailler sur un projet d'une certaine taille ne signifie pas forcément que l'on travaille sur un projet intéressant, mais ça augmente sensiblement les chances de rencontrer de nouveaux problèmes. Il n'est pas question aujourd'hui de définir ce qu'est un projet d'une certaine taille, ni même de démontrer le postulat précédent ; il s'agit plutôt de parler d'une des dernières problématiques auxquelles j'ai dû faire face :

Comment éviter que les **invocations dynamiques de méthodes via la réflexion** rendent les performances d'une application ou d'un module catastrophiques ?

Réponse : En minimisant l'utilisation de la réflexion. C'est ce que je vais tenter de développer à travers un exemple directement inspiré du projet réel, quoiqu'adapté pour les besoins de l'article. Le langage utilisé sera C# 3.0, mais rien n'empêche d'utiliser C# 2.0.

2.2 - Invocations dynamiques de méthodes

Le langage C# supporte la réflexion. Dès lors il est envisageable d'appeler dynamiquement des méthodes. La puissance de ce mécanisme est souvent contrebalancée par son coût. Prenons un exemple basique :

```
public void CallMethod(object target, string methodName)
{
    Type type = target.GetType();
    MethodInfo methodInfo = type.GetMethod(methodName);
    if (methodInfo != null)
    {
        methodInfo.Invoke(target, null);
    }
}
```

La fonction précédente est capable d'exécuter une méthode (sans valeur de retour ni paramètre) à partir de son nom, sur un objet donné. Cette implémentation est un désastre mais là n'est pas le vrai problème. Le vrai problème est inhérent aux **performances**. Essayons de nous concentrer sur ce qui coûte :

- L'appel à GetType pour récupérer le type de l'objet ne coûte quasiment rien. Nous pouvons considérer que la durée de son exécution est négligeable.

- L'appel à `GetMethod` pour récupérer les métadonnées relatives à la méthode peut également être ignoré. Attention toutefois : le coût peut ici varier en fonction de la surcharge choisie pour `GetMethod`, et du type sur lequel on l'appelle.
- L'appel à `Invoke` pour exécuter dynamiquement la méthode prend beaucoup de temps. En fait, c'est ce qui est le plus coûteux. Dans un simple test consistant à invoquer dynamiquement la méthode `Clone` sur une chaîne de caractères, la durée d'exécution s'est avérée être plus de 400 fois supérieure à celle nécessaire à un appel classique.

Mais relativisons. Si vous êtes dans un scénario impliquant une seule invocation dynamique, vous n'allez sans doute jamais avoir à vous préoccuper des performances. Qu'un appel de méthode vous coûte 2 microsecondes au lieu de 4 nanosecondes n'est pas forcément dramatique ; en vérité, ça l'est rarement dans la plupart des applications.

Les scénarios critiques sont ceux qui vous forcent à utiliser la réflexion et l'invocation dynamique pour des tâches **cruciales** et **récurrentes**. Les petites microsecondes font les grandes décennies, etc.

Essayons donc de se mettre en situation en prenant un exemple un peu plus sérieux.

2.3 - Il nous faut la réflexion

Admettons que le rôle d'un module clé de notre application soit de traiter un flux de données ; ces données étant des **DTO** quelconques. Une des tâches du module consiste à inspecter les propriétés de ces objets, ou plus précisément, à déterminer si la valeur courante de chaque propriété correspond à la valeur par défaut du type de la propriété.

Un test unitaire vaut 1000 mots. Voici celui qui valide (presque) le fonctionnement d'un composant capable d'effectuer la vérification dont nous venons de parler :

```
[Test]
public void ShouldBeAbleToFindDefaultValues()
{
    IDefaultValueTester tester = CreateDefaultValueTester();

    Assert.IsFalse(tester.IsDefaultValue(typeof(int), 12));
    Assert.IsTrue(tester.IsDefaultValue(typeof(int), 0));
    Assert.IsTrue(tester.IsDefaultValue(typeof(double?), null));
    Assert.IsTrue(tester.IsDefaultValue(typeof(string), null));
    Assert.IsFalse(tester.IsDefaultValue(typeof(string), "test"));
    Assert.IsFalse(tester.IsDefaultValue(typeof(DateTime), DateTime.Now));
    Assert.IsTrue(tester.IsDefaultValue(typeof(DateTime), new DateTime()));
}
```

L'interface `IDefaultValueTester` est simple :

```
public interface IDefaultValueTester
{
    bool IsDefaultValue(Type type, object value);
}
```

L'objet de cet article n'est pas vraiment de trouver un moyen de faire passer ce test. Il existe d'ailleurs une solution évidente, bien que peu élégante, qui consisterait à utiliser un switch-like. Mais il ne faut pas oublier qu'en .NET, un type peut être :

- Un type référence : class, interface.
- Un type valeur : type de base (int, double, bool, etc.), enum, struct.
- Un type **nullable** : implémentation de la classe générique `Nullable<T>` (int?, double?, bool?, etc.)

Dès lors, il est assez facile d'imaginer que le switch sera immonde, et pas forcément performant puisqu'il faudrait dans certains cas - je pense aux structures - instancier des objets #templates# servant de base aux comparaisons.

J'ai pensé à une solution différente, basée sur les generics. Il est possible qu'il y ait encore plus malin, mais celle-ci a pour avantage de constituer un bon support pour cet article :

- Elle nécessite que l'on recoure à l'invocation dynamique de méthode.
- Elle est lente, donc bonne candidate à l'optimisation.

Le principe consiste à tirer partie du mot clé **default** (capable de retourner la valeur par défaut d'un type) que l'on utilise souvent dans le contexte de la généricité. Voyons donc comment l'exploiter pour implémenter le c#ur de cette solution :

```
public static bool IsDefaultValue<T>(object value)
{
    var type = typeof(T);

    if (!type.IsValueType)
        return value == null;

    if (value != null)
        return value.Equals(default(T));

    if (type.IsGenericType && type.GetGenericTypeDefinition() == typeof(Nullable<>))
        return true;

    var message = "The value type '{0}' can't be null.";
    throw new ArgumentException(string.Format(message, type.Name));
}
```

Tout le monde aura noté la subtile différence qu'il y a entre le prototype de cette méthode et celui de l'unique méthode `IsDefaultValue` de l'interface `IDefaultValueTester` : le type qui était passé sous la forme d'un objet de type `Type` se retrouve à présent en paramètre de type `T` d'une méthode générique. Et c'est là qu'interviennent la réflexion et l'invocation dynamique de méthode.

Examinons donc l'implémentation de `IDefaultValueTester` que je propose :

```
public class DefaultValueTester : IDefaultValueTester
{
    public bool IsDefaultValue(Type type, object value)
    {
        // On récupère les métadonnées de la méthode générique statique
        // IsDefaultValue présente dans la classe courante.
        var thisType = typeof(DefaultValueTester);
        var methodInfo = thisType.GetMethod("IsDefaultValue",
            BindingFlags.Static
            | BindingFlags.Public);

        // On lui spécifie au runtime son paramètre de type générique
        methodInfo = methodInfo.MakeGenericMethod(type);

        // On invoque dynamiquement la méthode générique fermée
        return (bool) methodInfo.Invoke(null, new [] {value});
    }

    public static bool IsDefaultValue<T>(object value)
    {
        // cf. extrait de code précédent.
    }
}
```

La réflexion est donc nécessaire pour spécifier lors de l'exécution le paramètre de type de la méthode générique. Une fois la méthode fermée, on utilise Invoke pour l'exécuter dynamiquement.

Résultat : **le test passe**.

2.4 - Peut-on optimiser ?

Oui, et heureusement. Car si on s'en tient à la description du contexte, il serait assez irresponsable d'utiliser ce DefaultValueTester au niveau du composant clé de notre architecture. Rappelons que ce dernier est censé inspecter les propriétés de tous les objets arrivant sur le flux. Si nous considérons qu'il s'agit d'un flux faisant parvenir au module des milliers d'objets par seconde, l'optimisation n'est plus une option.

Voici un nouveau test qui nous permet d'avoir une idée à propos des performances de la solution actuelle :

```
[Test]
public void Test()
{
    var stopwatch = new Stopwatch();
    const int iterationCount = 100000;
    stopwatch.Start();

    IDefaultValueTester tester = CreateDefaultValueTester();
    for (var i = 0; i < iterationCount; i++)
    {
        tester.IsDefaultValue(typeof(int), 12);
        tester.IsDefaultValue(typeof(int), 0);
        tester.IsDefaultValue(typeof(double?), null);
        tester.IsDefaultValue(typeof(string), null);
        tester.IsDefaultValue(typeof(string), "test");
        tester.IsDefaultValue(typeof(DateTime), DateTime.Now);
        tester.IsDefaultValue(typeof(DateTime), new DateTime());
    }

    stopwatch.Stop();
    Console.WriteLine(string.Format("{0} iteration(s) in {1} ms.",
                                    iterationCount,
                                    stopwatch.ElapsedMilliseconds));
}
```

Les 100000 itérations ont été effectuées en **10625 ms**. C'est loin d'être terrible.

Il existe au moins deux façons différentes d'optimiser les invocations dynamiques de méthodes en C# dans un tel scénario. Les deux introduisent la notion de **caching** et se basent sur les **délégués**.

Première solution

- Utiliser la **Lightweight Code Generation (LCG)** via Reflection.Emit pour générer dynamiquement le code CIL correspondant à l'appel de la méthode générique IsDefaultValue<T> fermée sur le bon type.
- Utiliser la méthode CreateDelegate de DynamicMethod pour récupérer un délégué pointant sur la méthode que l'on vient de générer.

Seconde solution

- Conserver l'usage de la réflexion pour fermer le type de la méthode générique (appel à la méthode MakeGenericMethod) et récupérer le MethodInfo correspondant.
- Utiliser la méthode statique CreateDelegate de la classe abstraite Delegate pour récupérer un délégué à partir du MethodInfo précédent.

Les délégués ainsi obtenus ont deux particularités qui les rendent précieux dans le contexte de cette optimisation :

- Le coût de leur invocation est quasiment nul, contrairement à l'appel à la méthode Invoke sur un MethodInfo.

- Ils peuvent être mis en cache, et indexés intelligemment de façon à ce qu'ils ne soient pas recréés à chaque fois. Ici, il suffit d'utiliser le type des propriétés comme clé.

Retenons dans un premier la seconde solution, qui est plus simple à développer, maintenir et tester, et voyons ce que cela donne :

```
using TesterMethodDelegate = Func<object, bool>;

public class OptimizedDefaultValueTester : IDefaultValueTester
{
    private readonly Dictionary<Type, TesterMethodDelegate> cache = new Dictionary<Type,
TesterMethodDelegate>();

    public bool IsDefaultValue(Type type, object value)
    {
        var tester = GetTesterMethodDelegate(type, value);
        return tester(value);
    }

    private TesterMethodDelegate GetTesterMethodDelegate(Type type, object value)
    {
        TesterMethodDelegate tester;
        if(!this.cache.TryGetValue(type, out tester))
        {
            var thisType = typeof(OptimizedDefaultValueTester);
            var methodInfo = thisType.GetMethod("IsDefaultValue", BindingFlags.Static |
BindingFlags.Public);
            methodInfo = methodInfo.MakeGenericMethod(type);
            tester = (TesterMethodDelegate) Delegate.CreateDelegate(typeof(TesterMethodDelegate),
methodInfo);
            this.cache.Add(type, tester);
        }
        return tester;
    }

    public static bool IsDefaultValue<T>(object obj)
    {
        // cf. extraits de code précédents.
    }
}
```

Les points remarquables :

- **Ligne 1** : On utilise le type de délégué générique **Func<T,TReturn>** du Framework 3.5, derrière un alias (TesterMethodDelegate).
- **Ligne 5** : Les instances de TesterMethodDelegate sont mises en cache grâce à un champ d'instance de type Dictionary<TKey,TValue> et sont indexées par type.
- **Ligne 16** : A chaque appel à IsDefaultValue, on regarde si un TesterMethodDelegate a déjà été créé pour le type passé en paramètre. Si c'est le cas, on récupère l'instance dans le cache, sinon, on la crée avant de l'ajouter au cache.
- **Ligne 10** : L'appel dynamique via la méthode Invoke a disparu, on invoque directement l'instance de TesterMethodDelegate récupérée.

Toujours selon le même test, les performances sont améliorées. On passe de **10625 ms** pour 700000 appels, à **506 ms**. C'est environ 20 fois mieux, hurra.

2.5 - Peut-on aller plus loin ?

Pas vraiment, si on ne considère que les performances. Par contre, il est possible d'encapsuler le mécanisme d'optimisation précédent pour favoriser la réutilisabilité. Et pour cela, nous pouvons utiliser la **memoization**. Sans le savoir c'est un peu ce que nous avons imaginé jusque'ici.

Cependant, il est possible en utilisant les **méthodes anonymes** (ou les **expressions lambda** en C# 3.0) de mettre en place une solution plus élégante, et autorisant la réutilisation du caching comme s'il s'agissait en quelque sorte d'un **aspect**. J'ai découvert cela en tombant sur ce **post** et j'ai été séduit.

Le principe consiste à créer une méthode capable de retourner une version mémoisée d'un délégué. On peut même en faire une **méthode d'extension** générique en C# 3.0:

```
public static class Memoization
{
    public static Func<T, TResult> Memoize<T, TResult>(this Func<T, TResult> function)
    {
        var cache = new Dictionary<T, TResult>();
        var nullCache = default(TResult);
        var isNullCacheSet = false;
        return parameter =>
        {
            TResult value;

            if (parameter == null && isNullCacheSet)
            {
                return nullCache;
            }

            if (parameter == null)
            {
                nullCache = function(parameter);
                isNullCacheSet = true;
                return nullCache;
            }

            if (cache.TryGetValue(parameter, out value))
            {
                return value;
            }

            value = function(parameter);
            cache.Add(parameter, value);
            return value;
        };
    }
}
```

Décortiquons cette méthode :

- Elle prend en paramètre un délégué de type `Func<T, TResult>` et retourne un délégué du même type. Pour simplifier, on peut dire que la méthode d'extension prend en paramètre et retourne une fonction dont le prototype est le suivant : `TResult Function(T param)`
- La fonction retournée est construite via une expression lambda qui se charge d'encapsuler la logique de caching. Elle a pour rôle de mémoriser le résultat (de type `TResult`) de la fonction à chaque valeur différente de `T` pour laquelle on l'appelle.
- C'est la valeur sauvegardée qui est retournée lorsqu'elle est présente dans le cache. En effet, le résultat de la fonction pour un paramètre donné ayant déjà été déterminé, il ne sert à rien d'exécuter de nouveau la fonction avec le même paramètre.
- Dans cet exemple, l'expression lambda qui sert à créer la fonction retournée est une **closure**. C'est ici que réside toute l'ingéniosité de cette technique. Le dictionnaire est une variable locale à la méthode `Memoize` référencée par la méthode lambda, donc du point de vue de cette dernière l'état du cache sera conservé entre chaque appel.
- La méthode doit traiter un cas particulier : Si la valeur `null` est passée en argument de la fonction, il n'est plus possible d'utiliser un dictionnaire pour mettre en cache le résultat puisque les clés de ce dernier ne peuvent être nulles. Nous utilisons donc une variable spécialement dédiée : `nullCache`.

Bon, la memoization, c'est classe. Mais revenons à notre besoin. Quelle fonction a besoin d'être mémoisée ? Et bien tout simplement celle qui pour un type donné est capable de nous retourner le délégué pointant sur la bonne version fermée de `IsDefaultValue<T>`.

Une telle fonction peut avoir le prototype suivant :

```
Func<object, bool> GetTesterMethodDelegate(Type type);
```

Le type du délégué correspondant est le suivant :

```
Func<Type, Func<object, bool>>
```

En effet, il s'agit bien d'une fonction qui prend un type en argument, et qui retourne une autre fonction prenant un objet en argument et retournant un booléen.

Il nous reste plus qu'à examiner la nouvelle implémentation de `IDefaultValueTester` qui se base sur le principe :

```
using TesterMethodDelegate = Func<object, bool>;
using TesterMethodLocatorDelegate = Func<Type, Func<object, bool>>;

public class MemoizedDefaultValueTester : IDefaultValueTester
{
    private readonly TesterMethodLocatorDelegate testerLocator;

    public MemoizedDefaultValueTester()
    {
        // On crée une fonction capable de retourner la version fermée
        // de la méthode IsDefaultValue<T> pour un type donné.
        this.testerLocator = type =>
        {
            var thisType = typeof (OptimizedDefaultValueTester);
            var methodInfo = thisType.GetMethod("IsDefaultValue",
BindingFlags.Static | BindingFlags.Public);
            methodInfo = methodInfo.MakeGenericMethod(type);
            var tester =
(TesterMethodDelegate)Delegate.CreateDelegate(typeof (TesterMethodDelegate), methodInfo);
            return tester;
        };

        // On mémorise cette fonction en appelant notre méthode d'extension
        this.testerLocator = this.testerLocator.Memoize();
    }

    public bool IsDefaultValue(Type type, object value)
    {
        var tester = this.testerLocator(type);
        return tester(value);
    }

    public static bool IsDefaultValue<T>(object obj)
    {
        // cf. extraits de code précédents.
    }
}
```

Nous pouvons constater que :

- Un nouvel alias (`TesterMethodLocatorDelegate`) est introduit pour le type générique `Func<Type,Func<object,bool>>`
- La classe possède un champ d'instance (`testerLocator`) de type `TesterMethodLocatorDelegate` qui est initialisé dans le constructeur.

- La méthode d'extension Memoize définie plus tôt est utilisée pour mémoriser le TesterMethodLocatorDelegate, toujours au niveau du constructeur.
- L'implémentation de la méthode IsDefaultValue est extrêmement simplifiée. Via la version mémorisée du TesterMethodLocatorDelegate, on récupère un TesterMethodDelegate qui peut être invoqué directement afin d'effectuer le test sur la valeur passée en paramètre.

Les performances de cette solution sont les mêmes que celles mesurées pour la précédente : environ **510 ms**. Mais à présent, nous disposons d'une méthode Memoize pouvant être réutilisée.

Hum. Attendez# Pourquoi ne pas la réutiliser alors, pour mémoriser aussi les instances de TesterMethodDelegate retournées par le TesterMethodLocatorDelegate ? Il y aurait ainsi deux niveaux de mémorisation, et peut-être à la clé un petit gain de performance supplémentaire.

La modification se limite donc à ajouter un nouvel appel à la méthode Memoize dans le constructeur du MemoizedDefaultValueTester :

```
public MemoizedDefaultValueTester()
{
    // On crée une fonction capable de retourner la version fermée
    // de la méthode IsDefaultValue<T> pour un type donné.
    this.testeurLocator = type =>
    {
        var thisType = typeof (OptimizedDefaultValueTester);
        var methodInfo = thisType.GetMethod("IsDefaultValue",
BindingFlags.Static | BindingFlags.Public);
        methodInfo = methodInfo.MakeGenericMethod(type);
        var tester =
(TesterMethodDelegate)Delegate.CreateDelegate(typeof (TesterMethodDelegate), methodInfo);

        // On retourne à présent une version mémorisée du TesterMethodDelegate :
        return tester.Memoize();
    };

    // On mémorise cette fonction en appelant notre méthode d'extension
    this.testeurLocator = this.testeurLocator.Memoize();
}
```

Le test des performances indique à présent que les 100000 itérations ont eu lieu en **320 ms** ! Ce n'est pas aussi efficace que la première étape d'optimisation, mais proportionnellement cela conduit tout de même à une amélioration significative : environ 30%.



*Attention quand même : Si notre premier usage de la memoization était justifié, le second peut être très dangereux. **Nous gagnons des millisecondes, mais nous perdons des octets.** Et dans le scénario actuel, il est même probable que la memoization des TesterMethodDelegate conduise à une OutOfMemoryException rapidement...*

2.6 - Conclusion

Nous nous sommes contentés d'évoquer la solution d'optimisation impliquant la génération de code IL. Dans le contexte de la problématique discutée, elle n'offrait aucun avantage par rapport à celle que nous avons exposée. Pire, elle imposait une étape inutile. Toutefois, certains besoins plus complexes dépassent le cadre des invocations dynamiques de méthodes et peuvent tout de même être adressés efficacement en recourant à la génération de bytecode.

Quant à la technique détaillée dans celui-ci, elle est simple et permet d'obtenir des performances acceptables dans la majorité des scénarios pour lesquels l'utilisation de la réflexion est nécessaire.

Cela fait presque deux raisons de ne plus avoir systématiquement peur de la réflexion. Par contre, si vous n'en avez pas peur du tout, il serait peut-être temps de s'y mettre doucement...

3 - Lightweight Code Generation (LCG)

3.1 - Introduction

S'il existe des scénarios dans lesquels le recours à la LCG est inutile voire pénalisant, il n'y a parfois aucune autre alternative lorsqu'il s'agit de mettre en place une solution où les performances sont aussi importantes que la dynamique.

Vous ne trouverez pas dans cette partie un tutorial sur l'utilisation de `Reflection.Emit`, mais plutôt un exemple d'utilisation de cette technique pour répondre de façon optimale à un besoin bien spécifique. Nous essaierons en parallèle de faire ressortir quelques guidelines relatives à l'usage de la LCG.

3.2 - Mais c'est quoi ce truc ?

La LCG, ou Lightweight Code Generation, fait référence à une nouveauté apparue dans la seconde version du Framework .NET.

Il a toujours été possible en .NET d'utiliser l'API du namespace `System.Reflection.Emit` pour générer des assemblages, des modules et des types dynamiquement. Le principe est simple : on autorise via cette API les développeurs de la plateforme .NET à produire directement du code intermédiaire. C'est extrêmement puissant, mais très rapidement complexe. Le fait que le CIL ainsi obtenu soit (**quasiment**) impossible à déboguer ne vient pas arranger les choses.

C'était en quelque sorte la Heavyweight Code Generation : pour générer dynamiquement le **bytecode** correspondant à un traitement, il fallait la plupart du temps se taper la création d'un assembly, d'un module et d'un type pour finalement héberger la méthode encapsulant l'opération. Tout cela est souvent nécessaire, mais le reste du temps, c'est juste lourd.

Typiquement, lorsqu'on utilise la génération de CIL pour créer un proxy dynamiquement, on veut définir un type complet avec ses membres et ses méthodes. En revanche, pour créer une simple méthode au runtime on préférerait s'en passer.

La LCG autorise justement la création de méthodes dynamiques pouvant être réclamées par le garbage collector, et surtout ayant la capacité d'être hébergées anonymement, sans que l'on ait à créer d'assembly, de module ou de type. Outre le fait que de telles méthodes soient relativement faciles à créer (je ne parle pas de la génération de leurs corps), elles peuvent également être invoquées via des délégués. Et ça on connaît, c'est efficace.

3.3 - Parfois, c'est inutile

Dans l'article précédent, nous avons vu comment il était possible de créer un délégué pointant sur un `MethodInfo` pour considérablement optimiser les invocations dynamiques. L'exemple s'y prêtait bien : nous connaissions la signature de la méthode à appeler dynamiquement donc nous pouvions :

- Définir un type de délégué correspondant
- Solliciter la méthode `CreateDelegate` de la classe `Delegate`
- Invoquer le délégué ainsi récupéré

C'est vraiment ce qu'il faut retenir. **Il n'y a aucune raison d'utiliser la LCG si les signatures des méthodes à appeler dynamiquement sont connues** et qu'il est possible de définir les délégués correspondants.

3.4 - Un exemple moins ingrat

Vous aurez compris que lorsque la signature des méthodes à appeler dynamiquement n'est pas connue durant le design, il est impossible de déclarer un délégué correspondant à la méthode que l'on veut appeler. Il en découle que l'invocation via délégué est à oublier, et qu'il ne reste donc que la bonne vieille méthode Invoke sur le MethodInfo.

Oui, celle-la même qui ruine les performances.

A l'origine, je voulais trouver un exemple ni trop idiot ni trop complexe pour mettre ce cas en évidence et introduire la LCG. Finalement, je n'ai pas été capable de trouver quelque chose respectant cet équilibre. Vous aurez donc droit à un exemple vraiment simple et super idiot : le **cloneur**.

Commençons par en définir l'interface :

```
public interface ICloner
{
    object Clone(object toClone);
}
```

Les types respectant ce contrat devront fournir via la méthode Clone un service capable de retourner un **shallow clone** de l'objet passé en paramètre. Pour simplifier ici, nous ne considèrerons que les propriétés publiques des objets.

Ecrivons directement un test unitaire :

```
[Test]
public void Test()
{
    var cloner = GetCloner();
    var p = new Person
    {
        Id = 1,
        Firstname = "Romain",
        Lastname = "Verdier",
        BirthDate = new DateTime(1976, 03, 02),
        Height = 1.65
    };
    var p2 = cloner.Clone(p) as Person;
    Assert.AreNotEqual(p, p2);
    Assert.AreEqual(p.Id, p2.Id);
    Assert.AreEqual(p.Firstname, p2.Firstname);
    Assert.AreEqual(p.Lastname, p2.Lastname);
    Assert.AreEqual(p.BirthDate, p2.BirthDate);
    Assert.AreEqual(p.Height, p2.Height);
}
```

En utilisant simplement la réflexion, on peut proposer l'implémentation non optimisée suivante :

```
public class Cloner : ICloner
{
    private readonly Func<Type, Func<object, object>> clonerLocator;

    public Cloner()
    {
        this.clonerLocator = ((Func<Type, Func<object, object>>)GetCloner).Memoize();
    }

    public object Clone(object toClone)
    {
        var cloner = this.clonerLocator(toClone.GetType());
    }
}
```

```

        return cloner(toClone);
    }

    private Func<object, object> GetCloner(Type type)
    {
        var constructorInfo = type.GetConstructor(Type.EmptyTypes);
        if (constructorInfo == null)
        {
            throw new
ArgumentException(string.Format("'{0}' type doesn't have a default constructor.", type.Name));
        }

        return toClone =>
        {
            var clone = Activator.CreateInstance(type);
            var propertyInfos = type.GetProperties(BindingFlags.Instance | BindingFlags.Public);
            foreach (var propertyInfo in propertyInfos)
            {
                var setterInfo = propertyInfo.GetSetMethod();
                var getterInfo = propertyInfo.GetGetMethod();
                if (setterInfo != null && getterInfo != null)
                {
                    // Deux invocations dynamiques ont lieu ici sans que l'on puisse
                    // utiliser de délégués.
                    setterInfo.Invoke(clone, new object[] {getterInfo.Invoke(toClone, null)});
                }
            }
            return clone;
        }
    }
}

```

Notons à propos du code précédent :

- Peu de vérifications sont faites, c'est à la fois volontaire et mal.
- Nous n'utilisons pas GetValue et SetValue sur les PropertyInfo pour rendre les invocations dynamiques de méthodes explicites : ici, on fait deux invocations dynamiques par propriété. Une sur le getter, et une sur le setter.
- Ne pas connaître le type de chaque propriété à l'avance signifie que l'on ne connaît pas la signature des getters et des setters. Ne pas connaître la signature des getters et des setters nous empêche d'utiliser la méthode décrite dans l'article précédent.
- La memoization et l'utilisation d'un locateur ne servent qu'à faciliter la comparaison que l'on pourra faire avec la prochaine solution, puisqu'on ne peut pas utiliser de caching dans celle-ci.

Puisque vous aimez les chiffres comme tout le monde, j'ai créé un test de performances effectuant un million d'appels à la méthode de clonage. Et houlala, c'est lent : **100000 appels en plus de 26 secondes**.

En utilisant la LCG, il va être possible de générer une méthode encapsulant la logique de clonage relative à un type donné. On pourra également créer un délégué pour cette méthode, afin qu'elle puisse être invoquée sans que les performances ne soient dégradées. Ce délégué - et on rejoint ici le principe d'optimisation commun à toutes les solutions - pourra être mis en cache pour éviter que la méthode dynamique ne soit régénérée systématiquement.

Voyons ce que ça donne :

```

public class CilCloner : ICloner
{
    private readonly Func<Type, Func<object, object>> clonerLocator;

    public Cloner()
    {
        this.clonerLocator = ((Func<Type, Func<object, object>>) GetCloner).Memoize();
    }

    public object Clone(object toClone)

```

```

    {
        var cloner = this.clonerLocator(toClone.GetType());
        return cloner(toClone);
    }

    private Func<object, object> GetCloner(Type type)
    {
        var constructorInfo = type.GetConstructor(Type.EmptyTypes);
        if (constructorInfo == null)
        {
            throw new
ArgumentException(string.Format("'{0}' type doesn't have a default constructor.", type.Name));
        }

        var dynamicMethod = new DynamicMethod(string.Format("<{0}>DoClone", type.Name),
            typeof (object),
            new []{typeof (object)},
            this.GetType());

        var propertyInfos = type.GetProperties(BindingFlags.Instance | BindingFlags.Public);

        var gen = dynamicMethod.GetILGenerator();
        var local = gen.DeclareLocal(type);
        gen.Emit(OpCodes.Newobj, constructorInfo);
        gen.Emit(OpCodes.Stloc, local);
        foreach(var propertyInfo in propertyInfos)
        {
            var setterInfo = propertyInfo.GetSetMethod();
            var getterInfo = propertyInfo.GetGetMethod();
            if (setterInfo != null && getterInfo != null)
            {
                gen.Emit(OpCodes.Ldloc, local);
                gen.Emit(OpCodes.Ldarg_0);
                gen.Emit(OpCodes.Callvirt, getterInfo);
                gen.Emit(OpCodes.Callvirt, setterInfo);
            }
        }
        gen.Emit(OpCodes.Ldloc, local);
        gen.Emit(OpCodes.Ret);

        return (Func<object, object>) dynamicMethod.CreateDelegate(typeof (Func<object, object>));
    }
}

```

Décortiquons une fois de plus la solution, en gardant à l'esprit que le but de l'article n'est pas d'apprendre à coder en CIL :

- Le membre `clonerLocator` est gardé en champ d'instance afin que l'on puisse le mémoïser.
- Le constructeur s'occupe de cette tâche en faisant appel à la méthode d'extension `Memoize` dont vous pourrez (re)trouver la définition dans l'article précédent.
- La méthode `Clone`, correspondant à l'implémentation de l'interface `ICloner`, se contente de récupérer via le locateur un délégué capable d'effectuer le clone de l'objet passé en paramètre. Elle l'invoque directement ensuite.
- Le coeur de la solution réside donc dans la méthode `GetCloner` :
 - Une **guard clause** permet de vérifier si le type de l'objet à créer expose bien un constructeur par défaut.
 - Une `DynamicMethod` est créée. Il s'agit d'une méthode prenant en paramètre un `object` et retournant un `object`, qui sera capable de cloner un objet du type `type`
 - Grâce à la réflexion, on récupère les métadonnées de toutes les propriétés publiques du type `type`.
 - On récupère le générateur de code IL de la méthode dynamique afin de pouvoir émettre le corps de cette dernière.
 - On génère le code IL correspondant à la déclaration d'une variable locale de type `type`. On l'initialise avec une nouvelle instance de `type`.
 - On itère sur toutes les métadonnées des propriétés, et on ne considère que celles qui sont à la fois accessibles en lecture et en écriture.

- A chaque fois que cette condition est vérifiée, on produit le code correspondant à l'affectation de la propriété du clone (appel au setter). La valeur utilisée pour l'affectation est celle récupérée par le getter sur la propriété de l'objet à cloner.
- On termine la construction du corps de la DynamicMethod en émettant les instructions IL correspondant au retour du clone.
- Enfin, la méthode GetCloner crée un délégué à partir de la méthode dynamique via CreateDelegate, et le renvoie.

C'est beau ! La mesure des performances en utilisant le même test que précédemment indique cette fois qu'**un million de clones ont été créés en 305 ms**. Pour information, la méthode MemberwiseClone donne un résultat de **234 ms**.

3.5 - Conclusion

La Lightweight Code Generation, et plus globalement l'utilisation de Reflection.Emit, permet d'apporter des solutions insoupçonnées à certains problèmes bien spécifiques. Cependant, elle n'est pas gratuite et demande un investissement non négligeable de la part de ceux qui veulent la maîtriser ou bien même s'en servir ponctuellement. Les développements peuvent être nettement ralentis tandis que les phases de debug et de maintenance risquent de devenir critiques.

Il est donc surtout important de :

- Savoir que la technique existe.
- Savoir reconnaître les scénarios qui rendent son emploi envisageable.

Il existe assez peu de bonnes ressources permettant d'apprendre à maîtriser cette technique. Je suis en train de lire **CIL Programming: Under the Hood of .NET** de **Jason Bock**, sans être spécialement séduit. Je conseille aux plus curieux de commencer par la MSDN et Google, puis de consulter les sources de projets qui utilisent l'émission de bytecode. Le blog de **Jean-Baptiste Evain** est également riche en infos au sujet du CIL : il est l'auteur entre autres de **Mono.Cecil**.

Mais j'y reviendrai.