DOI 10.31891/2307-5732-2024-339-4-62
УДК 004.624

ПОЗУР МИХАЙЛО
Вінницький національний технічний університет
https://orcid.org/0009-0003-5225-2453
e-mail: mixalchik545@gmail.com
ВОЙТКО ВІКТОРІЯ
Вінницький національний технічний університет
https://orcid.org/0000-0002-3329-7256
e-mail: dekanfki@i.ua

МЕТОД ОПТИМІЗАЦІЇ РЕФЛЕКСІЇ В .NET ЗА РАХУНОК ВИКОРИСТАННЯ ДИНАМІЧНОЇ КОМПІЛЯЦІЇ ЛЯМБДА ФУНКЦІЙ

Розглянуто та проаналізовано методи оптимізації рефлексії в .NET. Основним методом оптимізації є використання динамічної компіляції лямбда функцій за рахунок використання інших методів метапрограмування. Одним із підходів, що дозволяють досягти необхідного результату, є використання компіляції Expression. Виклик скомпільованої лямбда функції є швидшим за виклик рефлексії, але сама компіляція займає певний час. Для цього необхідно зберігати делегат скомпільованої функції в пам'яті для можливості її повторного виклику. У ході експериментів визначено, що на версіях .NET7 і вище пошук в кеші та виклик такої функції сумарно займає стільки ж часу, як і виклик рефлексії. Таким чином, для покращення швидкодії методу необхідно спочатку оптимізувати структуру даних, що зберігає лямбда функції. Було використано більш ефективну з точки зору пошуку структуру даних. За результатами експериментів встановлено, що така оптимізація дозволила підвищити швидкодію методу. Нова структура даних дозволила використати реалізації хеш таблиць, оптимізованих під роботу з незмінюваними даними, що, в свою чергу, дозволило додатково підвищити швидкодію методу. Проте такий підхід є можливим для використання лише у версіях .NET8 та вище. Для оптимізації роботи методу у випадках послідовного виклику для одного й того ж типу даних було використано кешування інформації про останній тип даних, для якого використовувався метод. Проведено експерименти для визначення швидкодії розробленого методу у порівнянні з рефлексією та методами, що використовують схожий принцип. Для проведення експериментів було використано бібліотеку Benchmark.NET. За результатами експериментів встановлено, що розроблений метод має кращу швидкодію у порівнянні з рефлексією та аналогами.

Ключові слова: метапрограмування, .NET, C#, рефлексія, Expression.

POZUR MYKHAYLO, VOITKO VIKTORIIA
Vinnytsia National Technical University

REFLECTION OPTIMIZATION METHOD IN .NET USING DYNAMIC COMPILATION OF LAMBDA FUNCTIONS

Reflection optimization methods in .NET are analyzed. The main method of optimization is the use of dynamic compilation of lambda functions. One approach to achieve the desired result is to use Expression compilation. Calling a compiled lambda function is faster than calling reflection, but the compilation itself takes some time. To avoid this drawback, it is necessary to store the delegate of the compiled function in memory so that it can be called again. In the course of experiments, it was determined that on versions .NET7 and higher, searching in the cache and calling such a function takes the same amount of time as calling reflection. Thus, to improve the speed of the method, it is necessary to optimize the data structure that stores the lambda function. Two-level hash-table was used to optimize search and storage of lambda functions. According to the results of the experiments, it was established that such optimization made it possible to improve the performance of the method. The new data structure also made it possible to use implementations of hash tables optimized for working with unchanged data, which allowed to further increase the performance of the method. However, this approach is only possible for use in .NET8 and higher versions. To optimize the method in cases of successive calls for the same data type, caching of information about the last data type for which the method was called was used. Experiments were conducted to determine the speed of the developed method in comparison with reflection and methods using a similar principle. The Benchmark.NET library was used to conduct experiments. According to the results of the experiments, it was established that the developed method has a better speed compared to reflection and analogues.

Keywords: metaprogramming, .NET, C#, reflection, Expression.

Постановка проблеми у загальному вигляді та її зв'язок із важливими науковими чи практичними завданнями

При розробці програмного забезпечення на платформі .NET досить часто використовується механізм рефлексії (System.Reflestion) для узагальнення тих чи інших частин програмного коду [1]. Це дозволяє значно скоротити кількість коду, адже узагальнений код може використовуватися для роботи з великою кількістю типів даних.

Проте такий підхід має низку вагомих недоліків. Хоча рефлексія є доволі простою з точки зору її використання, проте сам механізм її роботи є доволі складним, що часто призводить до неочікуваної поведінки програми або до появи критичних помилок [2]. Іншим вагомим недоліком рефлексії є її вплив на швидкодію. Цей недолік є найбільш помітним при частих викликах рефлексії [3].

Таким чином, актуальною є розробка методів оптимізації рефлексії за рахунок використання інших засобів метапрограмування платформи .NET.

Аналіз досліджень та публікацій

У статті [4] розглянуто результати застосування Expression для реалізації рефлексії, орієнтованої на зчитування значення властивостей. Автор використав механізм динамічної компіляції Expression в лямбда

функції для реалізації такого функціоналу. У результаті було отримано аналогічний до рефлексії функціонал зчитування властивостей зі значно вищою швидкодією у порівнянні з рефлексією.

У статті [5] розглянуто метод оптимізації рефлексії за рахунок використання динамічної компіляції Expression у лямбда функцію та кешування вже скомпільованої функції.

Формулювання цілей статті

Метою роботи є розробка методу оптимізації рефлексії з використанням динамічної компіляції лямбда функцій та кешування, що дозволить підвищити швидкодію в задачах обробки даних.

Виклад основного матеріалу

Частим сценарієм використання рефлексії є зчитування та присвоєння значень полів і властивостей класів[1]. Це дозволяє динамічно отримувати значення властивостей і полів класу. Проте зчитування та присвоєння значень у такий спосіб є значно повільнішим [3]. Для оптимізації такого сценарію використовують динамічну компіляцію лямбда функцій у процесі роботи додатку [4, 5]. Одним з інструментів, що дозволяє досягти такого результату, є механізм виразів (Expression) платформи .NET [6]. Результатом компіляції виразу є делегат скомпільованої лямбда функції. Сам собою виклик скомпільованої лямбда функції через делегат є швидшим за використання рефлексії, проте все ще повільнішим за звичайний доступ до властивостей класу [3].

Проте використання виразів для оптимізації рефлексії має низку недоліків та проблем, що потребують рішення. Перш за все, сама компіляція виразу потребує значно більшої кількості ресурсів, ніж один виклик рефлексії, що робить недоцільним використання такого підходу для випадків, коли необхідно оптимізувати невелику кількість викликів рефлексії. Результати порівняння швидкодії для одного виклику наведено на рисунку1.

```
BenchmarkDotNet v0.13.12, Windows 10 (10.0.19045.4529/22H2/2022Update)
Intel Core i7-9750H CPU 2.60GHz, 1 CPU, 12 logical and 6 physical cores
.NET SDK 8.0.205
  [Host]     : .NET 8.0.5 (8.0.524.21615), X64 RyuJIT AVX2
  DefaultJob : .NET 8.0.5 (8.0.524.21615), X64 RyuJIT AVX2
```

Method	Mean	Error	StdDev
ReflectionGet	33.45 ns	0.637 ns	0.565 ns
ExpressionCompileAndGet	46,782.02 ns	785.484 ns	696.311 ns

Рис. 1. Порівняння швидкодії рефлексії та компіляції Expression для одного виклику

Таким чином, для того, щоб використання виразів було ефективним з точки зору швидкодії, необхідно мати можливість повторно використовувати вже скомпільований вираз. Для цього необхідно застосовувати кешування. Доцільним тут є використання статичного класу зі статичною змінною, що зберігатиме усі делегати скомпільованих функцій. У якості структури даних можна використовувати хеш таблицю, яка в .NET реалізується за допомогою Dictionary<TKey, TValue>.

Лямбда функції потрібно компілювати для всіх властивостей класу при першій спробі використання такого виразу. Це зробить перший виклик значно повільнішим, проте в подальшому всі інші виклики для будь-яких властивостей класу будуть мати вищу швидкодію. Код створення лямбда функцій зображено на рисунку 2.

```
var properties = type.GetProperties(BindingFlags.Public | BindingFlags.Instance);
var delegatesDict = new Dictionary<string, Func<object, object>>(StringComparer.Ordinal);

foreach (var prop in properties)
{
    if (!prop.CanRead)
        continue;

    var paramExpression = Expression.Parameter(typeof(object), "x");

    var instanceCast =
        !type.IsValueType ?
            Expression.TypeAs(paramExpression, type) :
            Expression.Convert(paramExpression, type);

    var getDelegate =
        Expression.Lambda<Func<object, object>>(
            Expression.Convert(
                Expression.Property(instanceCast, prop.Name),
                typeof(object)),
            paramExpression)
        .Compile();

    delegatesDict.Add(prop.Name, getDelegate);
}

delegatesDict.TrimExcess();
```

Рис. 2. Код створення лямбда функцій для зчитування значень властивостей класу

Не дивлячись на те, що виклик лямбда функції є значно швидшим за рефлексію [3, 4, 5], у ситуаціях, які є більш наближеними до реального використання такого підходу, різниця у швидкодії є мінімальною або швидкодія скомпільованого методу є нижчою за рефлексію (див. рисунок 3).

```
BenchmarkDotNet v0.13.12, Windows 10 (10.0.19045.4529/22H2/2022Update)
Intel Core i7-9750H CPU 2.60GHz, 1 CPU, 12 logical and 6 physical cores
.NET SDK 8.0.205
  [Host]     : .NET 8.0.5 (8.0.524.21615), X64 RyuJIT AVX2
  Job-FUSJBM : .NET 8.0.5 (8.0.524.21615), X64 RyuJIT AVX2

MaxIterationCount=15  MinIterationCount=10
```

Method	N	Mean	Error	StdDev
ExpressionGet	1000	51.83 µs	1.798 µs	1.682 µs
ReflectionGet	1000	41.86 µs	0.449 µs	0.297 µs

Рис. 3. Порівняння швидкодії скомпільованих виразів та рефлексії

Таким отриманим результатам є кілька пояснень. Перш за все, починаючи з версії .NET7, рефлексія стала на порядок швидшою за рахунок оптимізацій на рівні середовища виконання [7], тоді як усі попередні дослідження проводилися на версіях .NET6 та нижче[3, 4, 5]. Другим фактором є те, що пошук за хеш таблицею хоч і є швидким, проте за деяких умов він все ще повільніший, ніж виклик рефлексії. Тому в такому випадку доцільно оптимізувати структуру даних для зберігання делегатів. Для цього можна використати дворівневу хеш таблицю, де перший рівень – це тип даних, а другий рівень – назва властивості. Це дозволить пришвидшити пошук необхідної лямбда функції. Результати порівняння швидкодії звичайної та оптимізованої структури даних подано на рисунку 4.

```
BenchmarkDotNet v0.13.12, Windows 10 (10.0.19045.4529/22H2/2022Update)
Intel Core i7-9750H CPU 2.60GHz, 1 CPU, 12 logical and 6 physical cores
.NET SDK 8.0.205
  [Host]     : .NET 8.0.5 (8.0.524.21615), X64 RyuJIT AVX2
  Job-WXMJBS : .NET 8.0.5 (8.0.524.21615), X64 RyuJIT AVX2

MaxIterationCount=15  MinIterationCount=10
```

Method	N	Mean	Error	StdDev	Ratio	RatioSD
ExpressionBaseGet	1000	48.22 µs	1.499 µs	1.403 µs	1.22	0.07
Expression2LevelCacheGet	1000	30.42 µs	0.806 µs	0.673 µs	0.76	0.04
ReflectionGet	1000	39.76 µs	1.773 µs	1.658 µs	1.00	0.00

Рис. 4. Порівняння швидкодії звичайної та оптимізованої структури даних

Така оптимізація також сприяє кращому використанню пам’яті, оскільки хеш-таблиці другого рівня не мають потреби у додаванні чи видаленні даних, що дозволить вивільнити зарезервовану під розширення пам’ять.

Як видно з останнього порівняння, застосування лямбда функції є лише на 24% швидшим за використання рефлексії. Це пов’язано з тим, що пошук необхідної функції у кеші все ще займає певний час. Одним із варіантів оптимізації є «запам’ятовування» останнього типу даних, для якого здійснювався виклик лямбда функції. Така оптимізація дозволить пришвидшити роботу методу у випадках послідовного виклику для одного й того ж типу даних (див. рисунок 5).

Method	N	Mean	Error	StdDev	Ratio	RatioSD
ExpressionBaseGet	1000	47.28 µs	1.589 µs	1.486 µs	1.25	0.05
Expression2LevelCacheGet	1000	33.66 µs	1.602 µs	1.498 µs	0.89	0.05
Expression2LevelCacheWithLastTypeCacheGet	1000	20.30 µs	0.455 µs	0.403 µs	0.54	0.01
ReflectionGet	1000	38.00 µs	0.736 µs	0.532 µs	1.00	0.00

Рис. 5. Порівняння швидкодії методу з кешуванням останнього типу даних

З результатів експерименту можна зробити висновок, що у випадку послідовного виклику методу для одного й того ж типу даних можна отримати приріст швидкодії близько 45% у порівнянні з рефлексією.

Ще одним варіантом підвищення швидкодії методу може бути застосування спеціального варіанту хеш таблиці, оптимізованого для роботи з незмінними даними. Оскільки метод кешує всі лямбда функції відразу при першому виклику, то необхідність у розширенні хеш таблиці для закешованого типу даних відпадає. Це дозволить використовувати більш оптимальну структуру даних.

У .NET8 було реалізовано набір колекцій, оптимізованих для роботи з незмінними даними. Однією з таких колекцій є FrozenDictionary<TKey, TValue>, яка є аналогом хеш таблиці. Ця колекція не дозволяє додавати, видаляти чи змінювати дані, проте є більш оптимізованою в операціях пошуку та зчитування даних [8].

На рисунку 6 зображено результати експерименту з порівняння швидкодії методу, що використовує FrozenDictionary. Метод, що використовує таку структуру даних, є на 75% швидшим за рефлексію та на 50% швидшим за варіант методу, що використовує Dictionary. Варто зазначити, що використання такого підходу можливе лише на версіях .NET8 та вище.

Method	N	Mean	Error	StdDev	Ratio	RatioSD
ExpressionOptimizedGet	1000	21.084 µs	0.3240 µs	0.1928 µs	0.53	0.02
ExpressionOptimizedFrozenDictionaryGet	1000	9.627 µs	0.4673 µs	0.4371 µs	0.25	0.01
ReflectionGet	1000	39.220 µs	1.6851 µs	1.5762 µs	1.00	0.00

Рис. 6. Порівняння швидкодії методу з використанням FrozenDictionary

Після всіх описаних покращень метод набуває вигляду:

1. Створюється кеш для скомпільованих лямбда функцій, де перший рівень – це тип даних, а другий рівень – назва властивості.
2. При спробі зчитати чи присвоїти значення за допомогою методу перевіряється, чи тип даних, з яким проводиться операція, відповідає типу даних, над яким останній раз виконувалась операція.
3. Якщо тип даних відповідає останньому типу даних, то застосовується останній кеш для типу даних та продовжується виконання з кроку 10. Якщо ні, то перевіряється, чи є тип даних у загальному кеші.
4. Якщо тип даних є в загальному кеші, то виконання продовжується з кроку 9, якщо ні, то починається процес кешування.
5. Спочатку створюються відповідні вирази, що зчитують та присвоюють значення властивостям класу. Створені вирази компілюються в лямбда функції.
6. Делегати лямбда функцій додаються до хеш таблиці, де ключем є назва властивості, а значення – делегат.
7. Якщо метод виконується на версії .NET8 і вище, створена хеш таблиця трансформується в оптимізований під зчитування варіант (FrozenDictionary).
8. Отримана хеш таблиця додається до загального кешу, де ключем є тип даних, а значенням – клас з посиланнями створеної хеш таблиці.
9. Зчитується запис кешу для типу даних. Змінні, що зберігають останній тип даних та останній кеш типу даних, присвоюються у значення поточного типу даних та кешу типу даних.
10. З отриманого кешу типу даних зчитується та виконується відповідний делегат лямбда функції в залежності від типу операції та назви властивості.

На рисунку 7 наведено частину лістингу коду реалізації запропонованого методу.

```
public static object GetPropertyValue(this object obj, string propertyName)
{
    var type = obj.GetType();

    if(_lastType == type)
        return _lastChacheItem.GetPropertyValueFuncsFrozen[propertyName](obj);

    if (!_cache.TryGetValue(type, out var cacheItem))
    {
        CacheType(type);
        cacheItem = _cache[type];
    }

    _lastType = type;
    _lastChacheItem = cacheItem;

    return cacheItem.GetPropertyValueFuncsFrozen[propertyName](obj);
}
```

Рис. 7. Частина лістингу коду реалізації методу

За аналогією з функціоналом для зчитування значення властивостей (див. рисунок 7) було реалізовано функціонал присвоєння властивостей.

Було проведено низку експериментів для визначення швидкодії розробленого методу в порівнянні з рефлексією. Як і для попередніх експериментів, для цього використовувалась бібліотека Benchmark.NET [9]. Результати експериментів показано на рисунку 8.

Method	N	TestType	Mean	Error	StdDev
ExpressionGet	10000	SingleType	103.7 µs	5.36 µs	5.01 µs
ReflectionGet	10000	SingleType	396.6 µs	10.91 µs	9.67 µs
ExpressionSet	10000	SingleType	305.7 µs	7.69 µs	6.81 µs
ReflectionSet	10000	SingleType	642.8 µs	35.76 µs	33.45 µs
ExpressionGet	10000	MultipleTypes	256.4 µs	8.28 µs	7.34 µs
ReflectionGet	10000	MultipleTypes	485.5 µs	8.86 µs	6.40 µs
ExpressionSet	10000	MultipleTypes	356.9 µs	17.68 µs	16.54 µs
ReflectionSet	10000	MultipleTypes	648.1 µs	33.53 µs	31.37 µs

Рис. 8. Результати експериментів з визначення швидкодії розробленого методу

В умовах послідовних викликів на одному й тому ж типі даних (SingleType) швидкодія зчитування властивості була на 74% вищою у порівнянні з рефлексією. У випадках роботи з різними типами даних (MultipleTypes) різниця була меншою, адже метод виявився лише на 48% швидшим за рефлексію. Таким чином, можна говорити про покращення від 48% до 74% в операціях зчитування значень властивостей (Get). У випадку з присвоєнням значень (Set) при послідовних викликах швидкодія була на 53% кращою, а у випадках з викликом на різних типах даних – на 45%.

Варто зазначити, що в кеш методу перед усіма тестами завжди додавалося 550 різних типів даних, які сумарно містили понад 12000 делегатів лямбда функцій. Це було зроблено для того, щоб перевірити роботу методу в умовах, наближених до реальних.

Висновки

Використання динамічної компіляції лямбда функцій не є достатнім для оптимізації рефлексії, особливо на версіях .NET7 і вище. Оскільки для повноцінного використання методу потрібно мати можливість виклику в будь-якому місці програмного коду, то метод має забезпечувати можливість ефективного знаходження та збереження відповідних лямбда функцій. Без цього різниця у швидкодії між методом та рефлексією буде мінімальною, особливо після оптимізації рефлексії в .NET7.

З урахуванням вище зазначених чинників було розроблено метод, що використовує динамічну компіляцію лямбда функцій за допомогою Expression. Розроблений метод використовує систему кешування, оптимізовану під швидкий пошук. Таким чином, вдалося досягти підвищення швидкодії на 48-74% в задачах динамічного зчитування значень властивостей класів та на 45-53% – в задачах присвоєння значень.

Література

1. Ingebrigtsen E. Metaprogramming in C#: Automate your .NET development and simplify overcomplicated code / Einar Ingebrigtsen. – Birmingham, 2023. – 352 с. – (Packt Publishing).

2. Hazzard K. Metaprogramming in .NET / K. Hazzard, J. Brock. – New York, 2013. – 360 с. – (Manning).

3. Warren M. Why is reflection slow? [Електронний ресурс] / Matt Warren. – 2016. – Режим доступу до ресурсу: https://mattwarren.org/2016/12/14/Why-is-Reflection-slow/.

4. Zanid Haytam. C# Expression Trees: Property Getters. [Електронний ресурс] / Zanid Haytam. – 2020. – Режим доступу до ресурсу: https://blog.zhaytam.com/2020/11/17/expression-trees-property-getter/.

5. Ricardo Peres. Using Generated Methods Instead of Reflection. [Електронний ресурс] / Ricardo Peres. – 2022. – Режим доступу до ресурсу: https://weblogs.asp.net/ricardoperes/using-generated-methods-instead-of-reflection.

6. Expression trees [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/expression-trees/.

7. Performance Improvements in .NET 7. [Електронний ресурс] – Режим доступу до ресурсу: https://devblogs.microsoft.com/dotnet/performance_improvements_in_net_7/

8. Henrique S. Domareski. .NET 8 – Frozen Collections. [Електронний ресурс] / Henrique S. Domareski. – 2024. – Режим доступу до ресурсу: https://henriquesd.medium.com/net-8-frozen-collections-404c1d7c5240.

9. Benchmark.NET. [Електронний ресурс] – Режим доступу до ресурсу: https://benchmarkdotnet.org/index.html.

References

1 Ingebrigtsen E. Metaprogramming in C#: Automate your .NET development and simplify overcomplicated code / Einar Ingebrigtsen. – Birmingham, 2023. – 352 c. – (Packt Publishing).

2 Hazzard K. Metaprogramming in .NET / K. Hazzard, J. Brock. – New York, 2013. – 360 c. – (Manning).

3 Warren M. Why is reflection slow? [Elektronnij resurs] / Matt Warren. – 2016. – Rezhim dostupu do resursu: https://mattwarren.org/2016/12/14/Why-is-Reflection-slow/.

4 Zanid Haytam. C# Expression Trees: Property Getters. [Elektronnij resurs] / Zanid Haytam. – 2020. – Rezhim dostupu do resursu: https://blog.zhaytam.com/2020/11/17/expression-trees-property-getter/.

5 Ricardo Peres. Using Generated Methods Instead of Reflection. [Elektronnij resurs] / Ricardo Peres. – 2022. – Rezhim dostupu do resursu: https://weblogs.asp.net/ricardoperes/using-generated-methods-instead-of-reflection.

6 Expression trees [Elektronnij resurs] – Rezhim dostupu do resursu: https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/expression-trees/.

7 Performance Improvements in .NET 7. [Elektronnij resurs] – Rezhim dostupu do resursu: https://devblogs.microsoft.com/dotnet/performance_improvements_in_net_7/

8 Henrique S. Domareski. .NET 8 – Frozen Collections. [Elektronnij resurs] / Henrique S. Domareski. – 2024. – Rezhim dostupu do resursu: https://henriquesd.medium.com/net-8-frozen-collections-404c1d7c5240.

9 Benchmark.NET. [Elektronnij resurs] – Rezhim dostupu do resursu: https://benchmarkdotnet.org/index.html.