МЕТАПРОГРАММИРОВАНИЕ В C# С ИСПОЛЬЗОВАНИЕМ ДЕРЕВЬЕВ ВЫРАЖЕНИЙ

Войтович А. С., Ломако А. В.
Кафедра информационных технологий автоматизированных систем,
Белорусский государственный университет информатики и радиоэлектороники
Минск, Республика Беларусь
E-mail: voitovich@gmail.com, lavlot@bsuir.by

Анализируются деревья выражений на платформе .NET. Рассматриваются методы построения деревьев выражений, компиляция кода на этапе выполнения, использование деревьев выражений для повышения производительности.

Введение

Деревья выражений – малоизвестная, но очень интересная функция .NET. Большинство людей, знают их как средство получения структур объектно-реляционного отображения, но, несмотря на то, что это наиболее распространенный вариант использования, он не единственный. Дереаья выражений позволяюбт реализовать самый разнообразный функционал, включая генерацию кода, транспиляцию, метапрограммирование и многое другое.

Целью данной статьи является обзор того, что такое деревья выражений и как с ними работать, а также тестирование сценария использования выажений для оптимизации работы с информацией о типах и пространством имен System.Reflection.

I. Построение деревьев выражений

Когда дело доходит до языков программирования, выражение описывает некоторую операцию над данными, которая дает определенный результат. Это одна из основополагающих конструкций любого языка.

Хотя для людей, это очевидно, для программной интерпретации этого представления нам нужна специальная структура данных. Эту структуру данных называют деревом выражений.

В C# деревья выражений можно создавать двумя способами: мы можем создавать их напрямую через API, а затем скомпилировать их в инструкции времени выполнения, или мы можем дизассемблировать их из предоставленных лямбда-выражений.

Фреймворк предлагает нам API для построения деревьев выражений через класс Expression, расположенный в пространстве имен System.Linq.Expressions. Он предоставляет различные фабричные методы, которые можно использовать для создания выражений разных типов. Рассмотрим создание выражения на рисунке 1.

```
public Func<string, string?> ConstructGreetingFunction()
{
    var personNameParameter =
        Expression.Parameter(typeof(string), "personName");

    // Condition
    var isNullOrWhiteSpaceMethod = typeof(string)
        .GetMethod(nameof(string.IsNullOrWhiteSpace));

    var condition = Expression.Not(
        Expression.Call(isNullOrWhiteSpaceMethod, personNameParameter));

    // True clause
    var concatMethod = typeof(string)
        .GetMethod(nameof(string.Concat), new[] { typeof(string), typeof(string) });

    var trueClause = Expression.Call(
        concatMethod,
        Expression.Constant("Greetings, "),
        personNameParameter
    );

    // False clause
    var falseClause =
        Expression.Constant(null, typeof(string));

    var conditional =
        Expression.Condition(condition, trueClause, falseClause);

    var lambda =
        Expression.Lambda<Func<string, string?>>(conditional, personNameParameter);

    return lambda.Compile();
}
```

Рис. 1 – Пример создания выражения

Сначала мы вызываем Expression.Parameter() для создания выражения параметра. Мы сможем использовать его для получения значения, переданного определенному параметру.

После этого мы извлекаем тип для получения ссылки на метод string.IsNullOrWhiteSpace(). Мы используем Expression.Call() для создания выражения вызова метода, которое представляет собой вызов string.IsNullOrWhiteSpace() с параметром, разрешенным выражением, которое мы создали ранее. Чтобы выполнить логическую операцию «нет» над результатом, мы вызываем Expression.Not(), чтобы обернуть вызов метода.

Чтобы составить положительное предложение, мы создаем операцию «добавить» с помощью Expression.Add(). В качестве операндов мы предоставляем постоянное выражение для строки «Greetings» и выражение параметра, полученное ранее.

Затем для отрицательного случая мы используем Expression.Constant() для создания нулевого константного выражения. Чтобы гарантировать правильность ввода значения null, мы явно указываем тип в качестве второго параметра.

Наконец, мы объединяем все вышеперечисленные части вместе, чтобы создать наш тер-

нарный условный оператор. Если вы потратите немного времени на то, чтобы проследить, что входит в Expression.Condition(), вы поймете, что мы по существу воспроизвели древовидную диаграмму, которую видели ранее.

Однако само по себе это выражение бесполезно. Поскольку мы создали его сами, нас не особо интересует его структура – вместо этого мы должны получить выражение. Для этого нам нужно создать точку входа, обернув все в лямбда-выражение. Чтобы превратить его в настоящую лямбду, мы можем вызвать метод Compile(), который создаст делегат, который мы можем вызвать.

II. Повышение производительности с использованием выражений

Когда дело доходит до скомпилированных выражений, одним из наиболее распространенных сценариев использования является код с использованием reflection. Как известно, отражение может быть довольно медленным из-за позднего связывания, однако, компилируя код во время выполнения, мы можем добиться большей производительности.

Рассмотрим класс, на рисунке 2, у которого есть закрытый метод, который необходимо вызвать извне.

```
public class Command
{
    Ссылок: 0
    private int Execute() => 42;
}
```

Рис. 2 – Класс с закрытым методом

С помощью reflection это сделать довольно просто, рисунок 3:

```
public static int CallExecute(Command command) =>
    (int)typeof(Command)
        .GetMethod("Execute", BindingFlags.NonPublic | BindingFlags.Instance)
        .Invoke(command, null);
```

Рис. 3 – Вызов приватного метода с использованием reflection

Конечно, вызов такого метода может вызвать серьезные проблемы с производительностью, если мы поместим его в цикл. Прежде чем перейти к выражениям, мы можем сначала оптимизировать приведенный выше код, отделив часть, которая получает MethodInfo, от части, которая вызывает метод, поскольку если этот метод вызывается более одного раза, нам не нужно каждый раз использовать GetMethod(), рисунок 4:

```
public static class ReflectionCached
{
    Ссылок: 1
    private static MethodInfo ExecuteMethod { get; } = typeof(Command)
        .GetMethod("Execute", BindingFlags.NonPublic | BindingFlags.Instance);

    Ссылок: 1
    public static int CallExecute(Command command) =>
        (int)ExecuteMethod.Invoke(command, null);
}
```

Рис. 4 – Вызов приватного метода с использованием reflection и кешированием

Таким образом мы получили практически максимальную производительность, насколько это возможно с использованием reflection. Теперь сделаем то же самое, используя скомпилированное выражение, рисунок 5:

```
public static class ExpressionTrees
{
    Ссылок: 1
    private static MethodInfo ExecuteMethod { get; } = typeof(Command)
        .GetMethod("Execute", BindingFlags.NonPublic | BindingFlags.Instance);

    Ссылок: 2
    private static Func<Command, int> Impl { get; }

    Ссылок: 0
    static ExpressionTrees()
    {
        var instance = Expression.Parameter(typeof(Command));
        var call = Expression.Call(instance, ExecuteMethod);
        Impl = Expression.Lambda<Func<Command, int>>(call, instance)
                         .Compile();
    }

    Ссылок: 1
    public static int CallExecute(Command command) => Impl(command);
}
```

Рис. 5 – Вызов приватного метода с использованием Expression.Compile

Во всех этих подходах используются статические конструкторы для инициализации свойств ленивым и потокобезопасным способом. Это гарантирует, что вся тяжелая работа произойдет только один раз – при первом обращении к членам этих классов.

Теперь сравним эти методы друг с другом и оценим их производительность с помощью Benchmark.NET, таблица 1:

Таблица 1 – Результаты тестирования производительности

Метод	Среднее время выполнения, нс	Потребление памяти, Б
Reflection	113.225	48
Reflection (cached)	86.783	48
Expressions	2.716	24

Заключение

Использование деревьев выражений позволялет значительно улучшить гибкость системы и осуществлять кодогенерацию на этапе выполнения программы. При этом, из проведенного исследования видно, что скомпилированные выражения позволяют достичь производительностити, сравнимой с нативной компиляцией, при этом получив все преимущества генерации кода и снизить накладные расходы на использование reflection.

1. Don Syme. 2006. Leveraging. NET meta-programming components from F# integrated queries and interoperable heterogeneous execution – Proceedings of the 2006 workshop on ML, 2006.
2. Hazzard K. Metaprogramming in .NET / Hazzard K., Bock J. – New York : Manning, 2019. – 360 p.