ZloyChert
11 фев 2020 в 10:36

Деревья выражений в C# на примере нахождения производной (Expression Tree Visitor vs Pattern matching)

18 мин 45K

.NET*, C#*, ООП*

Доброго времени суток. Деревья выражений, особенно в сочетании с паттерном Visitor, всегда являлись довольно запутанной темой. Поэтому чем больше разнообразной информации по этой теме, чем больше примеров, тем легче интересующимся будет найти что-то, что им понятно и полезно.

Статья построена как обычно — начинается с концептуальных основ и определений и заканчивается примерами и способами использования. Оглавление ниже.

Ну и целью не является навязать определенное решение или сказать, что одно лучше другого. Выводы предлагаю делать самим с учетом всех нюансов в вашем случае. Я же выскажу свое мнение на своем примере.

Деревья Выражений

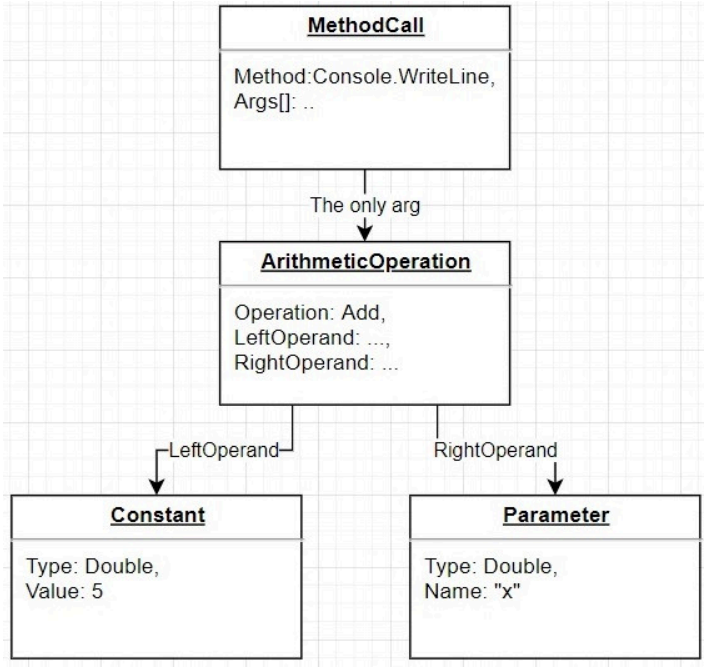
Основы

Для начала необходимо разобраться с деревьями выражений. Под ними понимают тип Expression или любой из его наследников (о них речь пойдет позже). При обычном раскладе выражение/алгоритмы представляются в виде выполняемого кода/инструкций, с которыми пользователь может не так и много чего делать (в основном выполнять). Тип Expression позволяет представить выражение/алгоритм (как правило лямбды, но не обязательно) как данные, организованные в виде древовидной структуры, к которой пользователь имеет доступ. Древовидный способ организации информации об алгоритме и название класса и дают нам «деревья выражений».

Для понятности разберем простой пример. Допустим, у нас есть лямбда

(x) => Console.WriteLine(x + 5)

Это можно представить в виде следующего дерева



Корень дерева — вершина "**MethodCall**", параметры метода — также выражения, следовательно может иметь сколько угодно потомков.

В нашем случае потомок один — вершина "**ArithmeticOperation**". В ней содержится информация о том, какая конкретно это операция и левый и правый операнды — также выражения. У такой вершины будет всегда 2 потомка.

Операнды представлены константой (**Constant**) и параметром (**Parameter**). У таких выражений нет потомков.

Это очень упрощенные примеры, но полностью отражающие суть.

Основная особенность деревьев выражений — это то, что их можно парсить и вычитывать всю необходимую информацию о том, что алгоритм должен делать. С какой-то точки зрения, это противоположность атрибутам. Атрибуты — средство декларативного описания поведения (очень условно, но конечная цель примерно такая). В то время как деревья выражений — использование функции/алгоритма для описания данных.

Используются они, например, в *провайдерах* entity framework. Применение очевидно — распарсить дерево выражений, понять, что там должно выполниться и составить по этому описанию *SQL*. Из менее известных примеров — библиотека для мокинга *moq*. Деревья выражений также нашли применение в *DLR* (dynamic language runtime). Разработчики компилятора используют их при обеспечении совместимости между динамической природой и dotnet, вместо генерации *MSIL*.

Также стоит упомянуть, что деревья выражений неизменяемы.

Синтаксис

Следующее, что стоит обговорить — синтаксис. Существуют 2 основных способа:

- Создание деревьев выражений через статические методы класса Expression
- Использование лямбда выражения, компилирующееся в Expression

Статические методы класса Expression

Создание деревьев выражений через статические методы класса Expression используется реже (в особенности с пользовательской точки зрения). Это громоздко, но довольно просто, в нашем распоряжении множество базовых кирпичиков, из которых можно построить довольно сложные вещи. Создание происходит через статические методы, т.к. конструкторы выражений имеют модификатор *internal*. И это не означает, что нужно расчехлять рефлексию.

В качестве примера привожу создание выражения с примера выше:

(x) => Console.WriteLine(x + 5)

```
ParameterExpression parameter = Expression.Parameter(typeof(double));
ConstantExpression constant = Expression.Constant(5d, typeof(double));
BinaryExpression add = Expression.Add(parameter, constant);
MethodInfo writeLine = typeof(Console).GetMethod(nameof(Console.WriteLine), new[] { typeof(doul
MethodCallExpression methodCall = Expression.Call(null, writeLine, add);
Expression<Action<double>> expressionlambda = Expression.Lambda<Action<double>>(methodCall, pa
Action<double> delegateLambda = expressionlambda.Compile();
delegateLambda(123321);
```

Может, это и не очень удобный способ, зато он полностью отражает внутреннюю структуру деревьев выражений. Плюс данный способ дает больше возможностей и фич, которые можно использовать в деревьях выражений: начиная от циклов, условий, try-catch, goto, присваивания, заканчивая блоками fault, отладочной информацией для точек останова, dynamic и тд.

Лямбда выражения

Использование лямбд в качестве выражений — более частый способ. Работает очень просто — умный компилятор на этапе компиляции смотрит, в качестве чего используется лямбда. И компилирует ее либо в делегат, либо в выражение. На уже освоенном примере, выглядит следующим образом

```
Expression<Action<double>> write = x => Console.WriteLine(x + 5);
```

Стоит уточнить такую вещь — выражение — это исчерпывающее описание. И его достаточно чтобы
получить результат. Деревья выражений типа LambdaExpression или его наследников могут быть сконвертировано в выполняемый IL. Остальные типы нельзя напрямую сконвертировать в выполняемы код (но это и не имеет много смысла).
Кстати, если кому-то критична быстрая компиляция выражения, можно взглянуть на этот сторонний проект.

Обратное же в общем случае неверно. Делегат не может просто так взять и представится выражением (но это все равно возможно).

Не все лямбды могут быть сконвертированы в деревья выражений. К таковым относятся:

- Содержащие оператор присваивания
- Содерщащие dynamic
- Асинхронные
- С телом (фигурные скобки)

```
double variable;
dynamic dynamic;
Expression<Action> assignment = () => variable = 5; //Compiler error: An expression tree may nc
Expression<Func<double>> dynamically = () => dynamic; //Compiler error: An expression tree may
Expression<Func<Task>> asynchon = async () => await Task.CompletedTask; //Compiler error: Asyn
Expression<Action> body = () => { }; //Compiler error: A lambda expression with a statement boc
```

Типы выражений

Предлагаю бегло ознакомится с доступными типами, чтобы представлять, какие возможности мы имеем. Все они находятся в пространстве имен System.Linq.Expressions

Я предлагаю для начала ознакомится с несколькими действительно интересными и необычными возможностями. Более простые типы выражений я собрал в табличку с кратким описанием.

Dynamic

С помощью DynamicExpression есть возможность использовать dynamic и все его фишки в деревьях выражений. Там довольно запутанное API, над этим примером я сидел дольше, чем над всеми остальными вместе взятыми. Вся запутанность обеспечивается кучей разного рода флагов. И некоторые из них похожи на те, которые вы ищите, но не обязательно ими являются. А при работе с dynamic в деревьях выражений говорящую ошибку получить сложно. Пример:

```
var parameter1 = Expression.Parameter(typeof(object), "name1");
var parameter2 = Expression.Parameter(typeof(object), "name2");
var dynamicParam1 = CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null);
var dynamicParam2 = CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null);
CallSiteBinder csb = Microsoft.CSharp.RuntimeBinder.Binder.BinaryOperation(CSharpBinderFlags.Nc
var dyno = Expression.Dynamic(csb, typeof(object), parameter1, parameter2);
Expression<Func<dynamic, dynamic, dynamic>> expr = Expression.Lambda<Func<dynamic, dynamic, dyn
Func<dynamic, dynamic, dynamic> action = expr.Compile();
var res = action("1", "2");
Console.WriteLine(res); //12
res = action(1, 2);
Console.WriteLine(res); //3
```

Я явно указал, откуда берется Binder во избежание путаницы с биндером из System.Reflection. Из интересностей — мы можем делать ref и out параметры, именованные параметры, унарные операции да и в принципе все, что можно сделать через dynamic, однако это потребует определенной сноровки.

Блоки перехвата исключений

Второе, на что обращу внимание — try/catch/finally/fault функциональность, а точнее на то, что у нас есть доступ к fault блоку. Он не доступен в C#, однако есть в MSIL.Это своеобразный аналог finally, который выполнится в случае любого исключения. В примере ниже выбросится исключение, после чего будет выведено «Hi» и программа будет ждать ввода. Только после этого она упадет окончательно. Не рекомендую эту практику к использованию.

```
var throwSmth = Expression.Throw(Expression.Constant(new Exception(), typeof(Exception)));
var log = Expression.Call(null, typeof(Console).GetMethod(nameof(Console.WriteLine), new[] { ty
var read = Expression.Call(null, typeof(Console).GetMethod(nameof(Console.ReadLine)));
var fault = Expression.TryFault(throwSmth, Expression.Block(new[] { log, read }));
Expression<Action> expr = Expression.Lambda<Action>(fault);
```

```
Action compiledExpression = expr.Compile();
compiledExpression();
```

▸ Краткое описание доступных типов деревьев выражений

Данных сведений достаточно, чтобы приступить к сравнению методов работы с деревьями выражений. Я решил разобрать это все на примере нахождения производной. Все возможные варианты я не предусматривал — лишь базовые. Но если кто-то зачем-то решил доработать и использовать — буду рад, если поделитесь улучшениями через реквест в мой репозиторий.

Сопоставление с образцом (pattern matching)

Итак, задача состоит в том, чтобы сделать тулу-считалку производных. Можно прикинуть следующее: есть пару правил нахождения производной для разных типов операции — умножение, деление и тд. В зависимости от операции необходимо выбрать определенную формулу. В такой банальной формулировке задача идеально ложиться на **switch/case**. А в последней версии языка нам представили switch/case 2.0 или **сопоставление с образцом (pattern matching)**.

Здесь сложно что-то обсуждать. На хабре такое количесво кода выглядит громоздко и плохо читается, поэтому предлагаю смотреть на github. На примере производной вышло так:

▸ Пример

Выглядит немного непривычно, но интересно. Писать такое было приятно — все условия органично ложатся в одну строку.

Пример говорит сам за себя, словами лучше и не опишешь.

Наивный visitor

В такой задаче тут же на ум приходит про expression tree visitor, который наводит много шума и чуть-чуть паники среди любителей обсудить аджайл на кухне. «Бойся ни незнания, а ложного знания. Лучше ничего не знать, чем считать правдой то, что неправда». Вспомнив сие чудную фразу Толстого, признав незнание и заручившись поддержкой гугла можно найти следующее руководство.

У меня данная ссылка является первой (после Сибири в 1949) по запросу «Expression tree visitor». На первый взгляд это именно то, что надо. Название статьи подходит к тому, что мы хотим сделать, а классы в примерах называются с суффиксом *Visitor*.

Ознакомившись со статьей и сделав по аналогии для нашего примера с производными получим: Ссылка на github.

▸ Пример

По сути — мы размазали свитч кейсы по разным классам. Меньше их не стало, магии не появилось. Все те же кейсы, гораздо больше строк. А где же обещанная двойная double dispatch диспетчеризация?

Классический visitor и двойная диспетчеризация

Здесь уже стоит рассказать про сам шаблон "**Посетитель**", он же **Visitor**, который и положен в основу *Expression tree visitor*'а. Разберем его как раз на примере деревьев выражений.

На секунду предположим, что мы разрабатываем деревья выражений. Мы хотим дать пользователям возможность итерироваться по дереву выражений и в зависимости от типов узлов (типов Expression'а) делать определенные действия.

Первый вариант — ничего не делать. То есть заставлять пользователей использовать switch/case. Это не такой и плохой вариант. Но здесь возникает такой нюанс: мы размазываем логику, отвечающую за конкретный тип. Говоря проще, полиморфизм и виртуальные вызовы (*ака позднее связывание*) дают возможность переложить определение типа на среду выполнения и убрать эти проверки из нашего кода. Нам достаточно иметь логику, которая создает экземпляр нужного типа, далее все будет сделано средой выполнения за нас.

Второй вариант. Очевидное решение — вынести логику в виртуальные методы. Переопределив виртуальный метод в каждом наследнике мы можем забыть о switch/case. Механизм полиморфных вызовов решит за нас. Здесь будет работать таблица методов, методы будут вызываться по смещению в ней. Но это тема на целую статью, так что не будем увлекаться. Кажется, что виртуальные методы решают нашу проблему. Но к сожалению, они создают другую. Для нашей задачи мы могли бы добавить метод GetDeriviative(). Но теперь сами классы выражений выглядят странно. Мы могли бы добавить таких методов на все случаи жизни, но они не вписываются под общую логику класса. Да и мы так и не предоставили возможность делать нечто подобное пользователям (адекватным путем, разумеется). Нам нужно дать пользователю определять логику для каждого конкретного типа, но сохранить полиморфизм (который доступен нам).

Одними лишь усилиями пользователя сделать это не удастся.

Вот тут и кроется настоящий визитор. В базовом типе иерархии (Expression в нашем случае) мы определим метод вида

```
virtual Expression Accept(ExpressionVisitor visitor);
```

В наследниках данный метод будет переопределен.

Сам ExpressionVisitor — базовый класс, содержащий по виртуальному методу с той же сигнатурой на каждый тип иерархии. На примере класса ExpressionVisitor — VisitBinary(...), VisitMethodCall(...), VisitConstant(...), VisitParameter(...).

Данные методы вызываются в соответствующем классе нашей иерархии.

Т.е. метод Accept в классе BinaryExpression будет выглядеть следующим образом:

```
protected internal override Expression Accept(ExpressionVisitor visitor)
{
        return visitor.VisitBinary(this);
}
```

В итоге для того, чтобы определить новое поведение, пользователю необходимо лишь создать наследника класса ExpressionVisitor, в котором будут переопределены соответствующие методы для решения одной задачи. В нашем случае создается DerivativeExpressionVisitor.

Далее мы имеем некие объекты наследников Expression, но каких — неизвестно, но и не надо. Мы вызываем виртуальный метод Accept с нужной нам реализацией ExpressionVisitor, т.е. с DerivativeExpressionVisitor. Благодаря динамической диспетчеризации вызовется переопределенная реализация Accept типа времени выполнения, скажем BinaryExpression. В теле этого метода мы прекрасно понимаем, что мы в BinaryExpression, но не знаем, какой именно наследник ExpressionVisitor к нам пришел. Но т.к. VisitBinary также виртуальный, нам не нужно знать. Опять же просто вызываем по ссылке на базовый класс, вызов динамически (во время

выполнения) диспетчеризуется и вызывается переопределенная реализация VisitBinary типа времени выполнения. Вот вам и двойная диспетчеризация — пинг-понг в стиле «ты выполняй» — «нет, ты».

Что нам это дает. Фактически, это дает возможность «добавлять» виртуальные методы извне, не изменяя класса. Звучит здорово, но имеет свои обратные стороны:

1. Какой-то левак в виде метода Accept, который отвечает за все и ни за что одновременно
2. Волновой эффект хорошей хэш-функции — при добавлении всего одного наследника в иерархию в худшем случае всем придется доделывать свои визиторы

Но природа деревьев выражений допускает данные издержки из-за специфики работы с выражениями, ведь данного рода обходы — одна из основных их особенностей.
Здесь можно посмотреть все методы доступные для перегрузки.

Итак, посмотрим как в итоге это выглядит.
Ссылка на github.

▸ Пример

Выводы

Пожалуй, как в большинстве задач программирования, однозначного ответа дать нельзя. Все, как всегда, зависит от конкретной ситуации. Мне нравится обычный паттерн матчинг для моего примера, т.к. я его не развил до масштабов промышленной разработки. В случае, если бы данное выражение бесконтролируемо увеличивалось, стоило бы задуматься о визиторе. И даже наивный визитор имеет право на жизнь — ведь это неплохой способ раскидать большое количество кода по классам, если иерархия не предоставила поддержку со своей стороны. И даже тут бывают исключения.

Точно также и сама поддержка визитора со стороны иерархии — очень спорная вещь.

Но я надеюсь, что предоставленной здесь информации хватит, чтобы сделать правильный выбор.

Теги: c#, c# 8, .net, pattern matching, oop, сопоставление с образцом, expression trees, expression tree visitor, expressions, деревья выражений

Хабы: .NET, C#, ООП

53 Карма 0 Рейтинг
Pavel Romash @ZloyChert
C# & .NET Monk
Подписаться