Expression Trees

Article • 03/09/2023

Expression trees represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

If you have used LINQ, you have experience with a rich library where the `Func` types are part of the API set. (If you aren't familiar with LINQ, you probably want to read the LINQ tutorial and the article about lambda expressions before this one.) Expression Trees provide richer interaction with the arguments that are functions.

You write function arguments, typically using Lambda Expressions, when you create LINQ queries. In a typical LINQ query, those function arguments are transformed into a delegate the compiler creates.

You've likely already written code that uses Expression trees. Entity Framework's LINQ APIs accept Expression trees as the arguments for the LINQ Query Expression Pattern. That enables Entity Framework to translate the query you wrote in C# into SQL that executes in the database engine. Another example is Moq, which is a popular mocking framework for .NET.

When you want to have a richer interaction, you need to use *Expression Trees*. Expression Trees represent code as a structure that you examine, modify, or execute. These tools give you the power to manipulate code during run time. You write code that examines running algorithms, or injects new capabilities. In more advanced scenarios, you modify running algorithms and even translate C# expressions into another form for execution in another environment.

You compile and run code represented by expression trees. Building and running expression trees enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see How to use expression trees to build dynamic queries.

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and .NET and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL). For more information about the DLR, see Dynamic Language Runtime Overview.

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the System.Linq.Expressions namespace.

When a lambda expression is assigned to a variable of type Expression<TDelegate>, the compiler emits code to build an expression tree that represents the lambda expression.

The C# compiler generates expression trees only from expression lambdas (or single-line lambdas). It can't parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in C#, see Lambda Expressions.

The following code examples demonstrate how to have the C# compiler create an expression tree that represents the lambda expression `num => num < 5`.

C#

```csharp
Expression<Func<int, bool>> lambda = num => num < 5;
```

You create expression trees in your code. You build the tree by creating each node and attaching the nodes into a tree structure. You learn how to create expressions in the article on building expression trees.

Expression trees are immutable. If you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You use an expression tree visitor to traverse the existing expression tree. For more information, see the article on translating expression trees.

Once you build an expression tree, you execute the code represented by the expression tree.

Limitations

There are some newer C# language elements that don't translate well into expression trees. Expression trees can't contain `await` expressions, or `async` lambda expressions. Many of the features added in C# 6 and later don't appear exactly as written in expression trees. Instead, newer features are exposed in expression trees in the equivalent, earlier syntax, where possible. Other constructs aren't available. It means that code that interprets expression trees works the same when new language features are introduced. However, the expression trees Even with these limitations, expression trees do enable you to create dynamic algorithms that rely on interpreting and modifying code that is represented as a data structure. It enables rich libraries such as Entity Framework to accomplish what they do.

Expression trees won't support new expression node types. It would be a breaking change for all libraries interpreting expression trees to introduce new node types. The following list includes most C# language elements that can't be used:

- Conditional methods that have been removed
- base access
- Method group expressions, including *address-of* (&) a method group, and anonymous method expressions
- References to local functions
- Statements, including assignment (`=`) and statement bodied expressions
- Partial methods with only a defining declaration
- Unsafe pointer operations
- dynamic operations
- Coalescing operators with null or default literal left side, null coalescing assignment, and the null propagating operator (?.)
- Multi-dimensional array initializers, indexed properties, and dictionary initializers
- throw expressions
- Accessing static virtual or abstract interface members
- Lambda expressions that have attributes
- Interpolated strings
- UTF-8 string conversions or UTF-8 string literals
- Method invocations using variable arguments, named arguments or optional arguments
- Expressions using System.Index or System.Range, index "from end" (^) operator or range expressions (..)
- async lambda expressions or await expressions, including await foreach and await using
- Tuple literals, tuple conversions, tuple == or !=, or with expressions
- Discards (_), deconstructing assignment, pattern matching is operator or the pattern matching switch expression
- COM call with `ref` omitted on the arguments
- ref, in or out parameters, `ref` return values, `out` arguments, or any values of ref struct type