Contents

Expression Trees (C#)

1/23/2019 • 4 minutes to read • Edit Online

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`.

You can compile and run code represented by expression trees. This 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 (C#).

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and the .NET Framework 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.

Creating Expression Trees from Lambda Expressions

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 can generate expression trees only from expression lambdas (or single-line lambdas). It cannot 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`.

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

Creating Expression Trees by Using the API

To create expression trees by using the API, use the Expression class. This class contains static factory methods that create expression tree nodes of specific types, for example, ParameterExpression, which represents a variable or parameter, or MethodCallExpression, which represents a method call. ParameterExpression, MethodCallExpression, and the other expression-specific types are also defined in the System.Linq.Expressions namespace. These types derive from the abstract type Expression.

The following code example demonstrates how to create an expression tree that represents the lambda expression `num => num < 5` by using the API.

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

In .NET Framework 4 or later, the expression trees API also supports assignments and control flow expressions such as loops, conditional blocks, and `try-catch` blocks. By using the API, you can create expression trees that are more complex than those that can be created from lambda expressions by the C# compiler. The following example demonstrates how to create an expression tree that calculates the factorial of a number.

```
// Creating a parameter expression.
ParameterExpression value = Expression.Parameter(typeof(int), "value");

// Creating an expression to hold a local variable.
ParameterExpression result = Expression.Parameter(typeof(int), "result");

// Creating a label to jump to from a loop.
LabelTarget label = Expression.Label(typeof(int));

// Creating a method body.
BlockExpression block = Expression.Block(
    // Adding a local variable.
    new[] { result },
    // Assigning a constant to a local variable: result = 1
    Expression.Assign(result, Expression.Constant(1)),
    // Adding a loop.
        Expression.Loop(
    // Adding a conditional block into the loop.
           Expression.IfThenElse(
    // Condition: value > 1
               Expression.GreaterThan(value, Expression.Constant(1)),
    // If true: result *= value --
               Expression.MultiplyAssign(result,
                   Expression.PostDecrementAssign(value)),
    // If false, exit the loop and go to the label.
               Expression.Break(label, result)
           ),
    // Label to jump to.
       label
    )
);

// Compile and execute an expression tree.
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()(5);

Console.WriteLine(factorial);
// Prints 120.
```

For more information, see Generating Dynamic Methods with Expression Trees in Visual Studio 2010, which also applies to later versions of Visual Studio.

Parsing Expression Trees

The following code example demonstrates how the expression tree that represents the lambda expression

`num => num < 5` can be decomposed into its parts.

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
                  param.Name, left.Name, operation.NodeType, right.Value);

// This code produces the following output:

// Decomposed expression: num => num LessThan 5
```

Immutability of Expression Trees

Expression trees should be immutable. This means that 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 can use an expression tree visitor to traverse the existing expression tree. For more information, see How to: Modify Expression Trees (C#).

Compiling Expression Trees

The Expression<TDelegate> type provides the Compile method that compiles the code represented by an expression tree into an executable delegate.

The following code example demonstrates how to compile an expression tree and run the resulting code.

```
// Creating an expression tree.
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));

// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

For more information, see How to: Execute Expression Trees (C#).

See also

- System.Linq.Expressions
- How to: Execute Expression Trees (C#)

- How to: Modify Expression Trees (C#)
- Lambda Expressions
- Dynamic Language Runtime Overview
- Programming Concepts (C#)

How to: Execute Expression Trees (C#)

1/23/2019 • 2 minutes to read • Edit Online

This topic shows you how to execute an expression tree. Executing an expression tree may return a value, or it may just perform an action such as calling a method.

Only expression trees that represent lambda expressions can be executed. Expression trees that represent lambda expressions are of type LambdaExpression or Expression<TDelegate>. To execute these expression trees, call the Compile method to create an executable delegate, and then invoke the delegate.

> **NOTE**
> If the type of the delegate is not known, that is, the lambda expression is of type LambdaExpression and not Expression<TDelegate>, you must call the DynamicInvoke method on the delegate instead of invoking it directly.

If an expression tree does not represent a lambda expression, you can create a new lambda expression that has the original expression tree as its body, by calling the Lambda<TDelegate>(Expression, IEnumerable<ParameterExpression>) method. Then, you can execute the lambda expression as described earlier in this section.

Example

The following code example demonstrates how to execute an expression tree that represents raising a number to a power by creating a lambda expression and executing it. The result, which represents the number raised to the power, is displayed.

```
// The expression tree to execute.
BinaryExpression be = Expression.Power(Expression.Constant(2D), Expression.Constant(3D));

// Create a lambda expression.
Expression<Func<double>> le = Expression.Lambda<Func<double>>(be);

// Compile the lambda expression.
Func<double> compiledExpression = le.Compile();

// Execute the lambda expression.
double result = compiledExpression();

// Display the result.
Console.WriteLine(result);

// This code produces the following output:
// 8
```

Compiling the Code

- Add a project reference to System.Core.dll if it is not already referenced.
- Include the System.Linq.Expressions namespace.

See also

- Expression Trees (C#)

- How to: Modify Expression Trees (C#)

How to: Modify Expression Trees (C#)

1/23/2019 • 2 minutes to read • Edit Online

This topic shows you how to modify an expression tree. Expression trees are immutable, which means that they cannot be modified directly. To change an expression tree, you must create a copy of an existing expression tree and when you create the copy, make the required changes. You can use the ExpressionVisitor class to traverse an existing expression tree and to copy each node that it visits.

To modify an expression tree

1. Create a new **Console Application** project.
2. Add a `using` directive to the file for the `System.Linq.Expressions` namespace.
3. Add the `AndAlsoModifier` class to your project.

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right, b.IsLiftedToNull,
b.Method);
        }

        return base.VisitBinary(b);
    }
}
```

   This class inherits the ExpressionVisitor class and is specialized to modify expressions that represent conditional `AND` operations. It changes these operations from a conditional `AND` to a conditional `OR`. To do this, the class overrides the VisitBinary method of the base type, because conditional `AND` expressions are represented as binary expressions. In the `VisitBinary` method, if the expression that is passed to it represents a conditional `AND` operation, the code constructs a new expression that contains the conditional `OR` operator instead of the conditional `AND` operator. If the expression that is passed to `VisitBinary` does not represent a conditional `AND` operation, the method defers to the base class implementation. The base class methods construct nodes that are like the expression trees that are passed in, but the nodes have their sub trees replaced with the expression trees that are produced recursively by the visitor.

4. Add a `using` directive to the file for the `System.Linq.Expressions` namespace.
5. Add code to the `Main` method in the Program.cs file to create an expression tree and pass it to the method that will modify it.

```
Expression<Func<string, bool>> expr = name => name.Length > 10 && name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression) expr);

Console.WriteLine(modifiedExpr);

/*  This code produces the following output:

    name => ((name.Length > 10) && name.StartsWith("G"))
    name => ((name.Length > 10) || name.StartsWith("G"))
*/
```

The code creates an expression that contains a conditional `AND` operation. It then creates an instance of the `AndAlsoModifier` class and passes the expression to the `Modify` method of this class. Both the original and the modified expression trees are outputted to show the change.

6. Compile and run the application.

See also

- How to: Execute Expression Trees (C#)
- Expression Trees (C#)

How to: Use Expression Trees to Build Dynamic Queries (C#)

1/23/2019 • 3 minutes to read • Edit Online

In LINQ, expression trees are used to represent structured queries that target sources of data that implement IQueryable<T>. For example, the LINQ provider implements the IQueryable<T> interface for querying relational data stores. The C# compiler compiles queries that target such data sources into code that builds an expression tree at runtime. The query provider can then traverse the expression tree data structure and translate it into a query language appropriate for the data source.

Expression trees are also used in LINQ to represent lambda expressions that are assigned to variables of type Expression<TDelegate>.

This topic describes how to use expression trees to create dynamic LINQ queries. Dynamic queries are useful when the specifics of a query are not known at compile time. For example, an application might provide a user interface that enables the end user to specify one or more predicates to filter the data. In order to use LINQ for querying, this kind of application must use expression trees to create the LINQ query at runtime.

Example

The following example shows you how to use expression trees to construct a query against an `IQueryable` data source and then execute it. The code builds an expression tree to represent the following query:

```
companies.Where(company => (company.ToLower() == "coho winery" || company.Length > 16)).OrderBy(company =>
company)
```

The factory methods in the System.Linq.Expressions namespace are used to create expression trees that represent the expressions that make up the overall query. The expressions that represent calls to the standard query operator methods refer to the Queryable implementations of these methods. The final expression tree is passed to the CreateQuery<TElement>(Expression) implementation of the provider of the `IQueryable` data source to create an executable query of type `IQueryable`. The results are obtained by enumerating that query variable.

```
// Add a using directive for System.Linq.Expressions.

string[] companies = { "Consolidated Messenger", "Alpine Ski House", "Southridge Video", "City Power & Light",
                   "Coho Winery", "Wide World Importers", "Graphic Design Institute", "Adventure Works",
                   "Humongous Insurance", "Woodgrove Bank", "Margie's Travel", "Northwind Traders",
                   "Blue Yonder Airlines", "Trey Research", "The Phone Company",
                   "Wingtip Toys", "Lucerne Publishing", "Fourth Coffee" };

// The IQueryable data to query.
IQueryable<String> queryableData = companies.AsQueryable<string>();

// Compose the expression tree that represents the parameter to the predicate.
ParameterExpression pe = Expression.Parameter(typeof(string), "company");

// ***** Where(company => (company.ToLower() == "coho winery" || company.Length > 16)) *****
// Create an expression tree that represents the expression 'company.ToLower() == "coho winery"'.
Expression left = Expression.Call(pe, typeof(string).GetMethod("ToLower", System.Type.EmptyTypes));
Expression right = Expression.Constant("coho winery");
Expression e1 = Expression.Equal(left, right);

// Create an expression tree that represents the expression 'company.Length > 16'.
left = Expression.Property(pe, typeof(string).GetProperty("Length"));
right = Expression.Constant(16, typeof(int));
Expression e2 = Expression.GreaterThan(left, right);
```

```
Expression e2 = Expression.GreaterThan(left, right);

// Combine the expression trees to create an expression tree that represents the
// expression '(company.ToLower() == "coho winery" || company.Length > 16)'.
Expression predicateBody = Expression.OrElse(e1, e2);

// Create an expression tree that represents the expression
// 'queryableData.Where(company => (company.ToLower() == "coho winery" || company.Length > 16))'
MethodCallExpression whereCallExpression = Expression.Call(
    typeof(Queryable),
    "Where",
    new Type[] { queryableData.ElementType },
    queryableData.Expression,
    Expression.Lambda<Func<string, bool>>(predicateBody, new ParameterExpression[] { pe }));
// ***** End Where *****

// ***** OrderBy(company => company) *****
// Create an expression tree that represents the expression
// 'whereCallExpression.OrderBy(company => company)'
MethodCallExpression orderByCallExpression = Expression.Call(
    typeof(Queryable),
    "OrderBy",
    new Type[] { queryableData.ElementType, queryableData.ElementType },
    whereCallExpression,
    Expression.Lambda<Func<string, string>>(pe, new ParameterExpression[] { pe }));
// ***** End OrderBy *****

// Create an executable query from the expression tree.
IQueryable<string> results = queryableData.Provider.CreateQuery<string>(orderByCallExpression);

// Enumerate the results.
foreach (string company in results)
    Console.WriteLine(company);

/*  This code produces the following output:

    Blue Yonder Airlines
    City Power & Light
    Coho Winery
    Consolidated Messenger
    Graphic Design Institute
    Humongous Insurance
    Lucerne Publishing
    Northwind Traders
    The Phone Company
    Wide World Importers
*/
```

This code uses a fixed number of expressions in the predicate that is passed to the `Queryable.Where` method. However, you can write an application that combines a variable number of predicate expressions that depends on the user input. You can also vary the standard query operators that are called in the query, depending on the input from the user.

Compiling the Code

- Create a new **Console Application** project.
- Add a reference to System.Core.dll if it is not already referenced.
- Include the System.Linq.Expressions namespace.
- Copy the code from the example and paste it into the `Main` method.

See also

- Expression Trees (C#)
- How to: Execute Expression Trees (C#)
- How to: Dynamically Specify Predicate Filters at Runtime

Debugging Expression Trees in Visual Studio (C#)

1/23/2019 • 2 minutes to read • Edit Online

You can analyze the structure and content of expression trees when you debug your applications. To get a quick overview of the expression tree structure, you can use the `DebugView` property, which is available only in debug mode. For more information about debugging, see Debugging in Visual Studio.

To better represent the content of expression trees, the `DebugView` property uses Visual Studio visualizers. For more information, see Create Custom Visualizers.

To open a visualizer for an expression tree

1. Click the magnifying glass icon that appears next to the `DebugView` property of an expression tree in **DataTips**, a **Watch** window, the **Autos** window, or the **Locals** window.

   A list of visualizers is displayed.

2. Click the visualizer you want to use.

Each expression type is displayed in the visualizer as described in the following sections.

ParameterExpressions

ParameterExpression variable names are displayed with a "$" symbol at the beginning.

If a parameter does not have a name, it is assigned an automatically generated name, such as `$var1` or `$var2`.

Examples

EXPRESSION	`DEBUGVIEW` PROPERTY
`ParameterExpression numParam = Expression.Parameter(typeof(int), "num");`	`$num`
`ParameterExpression numParam = Expression.Parameter(typeof(int));`	`$var1`

ConstantExpressions

For ConstantExpression objects that represent integer values, strings, and `null`, the value of the constant is displayed.

For numeric types that have standard suffixes as C# literals, the suffix is added to the value. The following table shows the suffixes associated with various numeric types.

TYPE	SUFFIX
UInt32	U
Int64	L
UInt64	UL

TYPE	SUFFIX
Double	D
Single	F
Decimal	M

Examples

EXPRESSION	`DEBUGVIEW` PROPERTY
`int num = 10; ConstantExpression expr = Expression.Constant(num);`	10
`double num = 10; ConstantExpression expr = Expression.Constant(num);`	10D

BlockExpression

If the type of a BlockExpression object differs from the type of the last expression in the block, the type is displayed in the `DebugInfo` property in angle brackets (< and >). Otherwise, the type of the BlockExpression object is not displayed.

Examples

EXPRESSION	`DEBUGVIEW` PROPERTY

`BlockExpression block = Expression.Block(Expression.Constant("test"));`	`.Block() {`
	`"test"`
	`}`

`BlockExpression block = Expression.Block(typeof(Object), Expression.Constant("test"));`	`.Block<System.Object>() {`
	`"test"`
	`}`

LambdaExpression

LambdaExpression objects are displayed together with their delegate types.

If a lambda expression does not have a name, it is assigned an automatically generated name, such as `#Lambda1` or `#Lambda2`.

Examples

EXPRESSION	`DEBUGVIEW` PROPERTY

EXPRESSION	DEBUGVIEW PROPERTY

`LambdaExpression lambda = Expression.Lambda<Func<int>>(Expression.Constant(1));`	`.Lambda #Lambda1<System.Func'1[System.Int32]>() {`
	`1`
	`}`

`LambdaExpression lambda = Expression.Lambda<Func<int>>(Expression.Constant(1), "SampleLambda", null);`	`.Lambda SampleLambda<System.Func'1[System.Int32]>() {`
	`1`
	`}`

LabelExpression

If you specify a default value for the LabelExpression object, this value is displayed before the LabelTarget object.

The `.Label` token indicates the start of the label. The `.LabelTarget` token indicates the destination of the target to jump to.

If a label does not have a name, it is assigned an automatically generated name, such as `#Label1` or `#Label2`.

Examples

EXPRESSION	DEBUGVIEW PROPERTY

`LabelTarget target = Expression.Label(typeof(int), "SampleLabel"); BlockExpression block = Expression.Block( Expression.Goto(target, Expression.Constant(0)), Expression.Label(target, Expression.Constant(-1)));`	`.Block() {`
	`.Goto SampleLabel { 0 };`
	`.Label`
	`-1`
	`.LabelTarget SampleLabel:`
	`}`

`LabelTarget target = Expression.Label(); BlockExpression block = Expression.Block( Expression.Goto(target5), Expression.Label(target5));`	`.Block() {`
	`.Goto #Label1 { };`
	`.Label`
	`.LabelTarget #Label1:`
	`}`

Checked Operators

Checked operators are displayed with the "#" symbol in front of the operator. For example, the checked addition operator is displayed as `#+`.

Examples

EXPRESSION	`DEBUGVIEW` PROPERTY
`Expression expr = Expression.AddChecked(` `Expression.Constant(1), Expression.Constant(2));`	`1 #+ 2`
`Expression expr = Expression.ConvertChecked(` `Expression.Constant(10.0), typeof(int));`	`#(System.Int32)10D`

See also

- Expression Trees (C#)
- Debugging in Visual Studio
- Create Custom Visualizers