Inhaltsverzeichnis

Searching a Tree of Objects with Linq, Revisited 1
Two types of Tree Traversal 1
Tree to IEnumerable<T> Extension methods 2

Searching a Tree of Objects with Linq, Revisited

A while back, I wrote about searching through a tree using linq to objects. That post was mostly snippets of code about delegates, lambda's, yield and how it applies to linq — more a technical exploration than an example. So I thought I'd follow it up with concrete extension methods to make virtually any tree searchable by Linq.

Linq, IEnumerable<T>, yield

All that is required to search a tree with Linq is creating a list of all nodes in the tree. Linq to Objects can operate on IEnumerable<T>. Really, **Linq to objects is a way of expressing operations we've been doing forever in loops with if/else blocks**. That means there isn't any search magic going on, it is a linear traversal of all elements in a set and examining each to determine whether it matches our search criteria.

To turn a tree into a list of node we need to walk and collect all children of every node. A simple task for a recursive list that carries along a list object to stuff every found node into. But there is a better way, using yield to return each item as it is encountered. Now we don't have to carry along a collection. **Iterators using yield implement a pattern in which a method can return more than once**. For this reason, a method using yield in C# must return an IEnumerable, so that the caller gets a handle to an object it can traverse the result of the multiple return values.

IEnumerable is basically an unbounded set. This is also the reason why unlike collections, it does not have a Count Property. It is entirely possible for an enumerator to return an infinite series of items.

Together IEnumerable<T> and yield are a perfect match for our problem, i.e. recursively walking a tree of nodes and return an unknown number of nodes.

Two types of Tree Traversal

Depth First

In depth-first traversal, the algorithm will dig continue to dig down a nodes children until it reaches a leaf node (a node without children), before considering the next child of the current parent node.

Breadth First

In breadth-first traversal, the algorithm will return all nodes at a particular depth first before considering the children at the next level. I.e. First return all the nodes from level 1, then all nodes from level 2, etc.

Tree to IEnumerable<T> Extension methods

```
1.     public static class TreeToEnumerableEx
2.     {
3.
4.      public static IEnumerable<T> AsDepthFirstEnumerable<T>(this T
   head, Func<T, IEnumerable<T>> childrenFunc)
5.      {
6.       yield return head;
7.       foreach (var node in childrenFunc(head))
8.       {
9.        foreach (var child in AsDepthFirstEnumerable(node,
   childrenFunc))
10.        {
11.         yield return child;
12.        }
13.       }
14.      }
15.
16.      public static IEnumerable<T> AsBreadthFirstEnumerable<T>(this T
   head, Func<T, IEnumerable<T>> childrenFunc)
17.      {
18.       yield return head;
19.       var last = head;
20.       foreach(var node in AsBreadthFirstEnumerable(head,childrenFunc))
21.       {
22.        foreach(var child in childrenFunc(node))
23.        {
24.         yield return child;
25.         last = child;
26.        }
27.        if(last.Equals(node)) yield break;
28.       }
29.      }
30.     }
```

This static class provides two extension methods that can be used on any object, as long as it's possible to express a function that returns all children of that object, i.e. the object is a node in some type of tree and has a method or property for accessing a list of its children.

An Example

Let's use a hypothetical Tree model defined by this Node class:

```
1.     public class Node
2.     {
```

```
3.      private readonly List<Node> children = new List<Node>();
4.
5.      public Node(int id)
6.      {
7.       Id = id;
8.      }
9.
10.     public IEnumerable<Node> Children { get { return children; } }
11.
12.     public Node AddChild(int id)
13.     {
14.      var child = new Node(id);
15.      children.Add(child);
16.      return child;
17.     }
18.
19.     public int Id { get; private set; }
20.    }
```

Each node simply contains a list of children and has an Id, so that we know what node we're looking at. The AddChild() method is a convenience method so we don't expose the child collection and no node can ever be added as a child twice.

The calling convention for a depth-first collection is:

```
1. IEnumerable<Node> = node.AsDepthFirstEnumerable(n => n.Children);
```

The lambda expression **n ⇒ n.Children** is the function that will return the children of a node. It simply states given n, return the value of the Children property of n. A simple test to verify that our extension works and to show us using the extension in linq looks like this:

```
1.     [Test]
2.     public void DepthFirst()
3.     {
4.      // build the tree in depth-first order
5.      int id = 1;
6.      var depthFirst = new Node(id);
7.      var df2 = depthFirst.AddChild(++id);
8.      var df3 = df2.AddChild(++id);
9.      var df4 = df2.AddChild(++id);
10.     var df5 = depthFirst.AddChild(++id);
11.     var df6 = df5.AddChild(++id);
12.     var df7 = df5.AddChild(++id);
13.
14.     // find all nodes in depth-first order and select just the Id of
    each node
15.     var IDs = from node in depthFirst.AsDepthFirstEnumerable(x =>
    x.Children)
```

```
16.             select node.Id;
17.
18.      // confirm that this list of IDs is in depth-first order
19.      Assert.AreEqual(new int[] { 1, 2, 3, 4, 5, 6, 7 }, IDs.ToArray());
20.     }
```

For breadth-first collections, the calling convention is:

```
1. IEnumerable<Node> = node.AsBreadthFirstEnumerable(n => n.Children);
```

Again, we can test that the extension works like this:

```
1.     [Test]
2.     public void BreadthFirst()
3.     {
4.      // build the tree in breadth-first order
5.      var id = 1;
6.      var breadthFirst = new Node(id);
7.      var bf2 = breadthFirst.AddChild(++id);
8.      var bf3 = breadthFirst.AddChild(++id);
9.      var bf4 = bf2.AddChild(++id);
10.     var bf5 = bf2.AddChild(++id);
11.     var bf6 = bf3.AddChild(++id);
12.     var bf7 = bf3.AddChild(++id);
13.
14.     // find all nodes in breadth-first order and select just the Id of
    each node
15.     var IDs = from node in breadthFirst.AsBreadthFirstEnumerable(x =>
    x.Children)
16.           select node.Id;
17.
18.     // confirm that this list of IDs is in depth-first order
19.     Assert.AreEqual(new int[] { 1, 2, 3, 4, 5, 6, 7 }, IDs.ToArray());
20.    }
```

Searching Trees

The tree used in the example is of course extremely simple, i.e. it doesn't even have any worthwhile data to query attached to a node. But these extension methods could be used on a node of any kind of tree, **allowing the full power of Linq, grouping, aggregation, sorting, projection, etc. to be used on the tree**.

As a final note, you may wonder, **why bother with depth-first vs. breadth first?** After all, in the end we do examine every node! There is however one particular case where the choice of algorithm can be very important: **You are looking for one match or a particular number of matches**. Since we are using yield, we can terminate the traversal at any time. Using the **FirstOrDefault()**

extension on our Linq expression, the traversal would stop as soon as one match is found. And if have any knowledge where that node might be in the tree, the choice of search algorithm can be a significant performance factor.

Link to the original post from ILogable

From:
https://jmz-elektronik.ch/dokuwiki/ - **Bücher & Dokumente**

Permanent link:
https://jmz-elektronik.ch/dokuwiki/doku.php?id=start:visualstudio2017:programmieren:dotnetgrundlagen:tipps_tricks:searchingtreeofobjectswithlinq

Last update: **2022/06/22 10:31**