C#中的Lambda表达式和表达式树

2015/03/01 · IT技术 · 2 评论 · .Net, C语言, lambda

分享到： 9

基于SSH实现员工管理系统之案例实现篇

Sass和Compass必备技能之Sass篇

less即学即用

vuejs入门基础

原文出处： 田小计划 的博客

在C# 2.0中，通过方法组转换和匿名方法，使委托的实现得到了极大的简化。但是，匿名方法仍然有些臃肿，而且当代码中充满了匿名方法的时候，可读性可能就会受到影响。C# 3.0中出现的Lambda表达式在不牺牲可读性的前提下，进一步简化了委托。

LINQ的基本功能就是创建操作管道，以及这些操作需要的任何状态。这些操作表示了各种关于数据的逻辑，例如数据筛选，数据排序等等。通常这些操作都是用委托来表示。Lambda表达式是对LINQ数据操作的一种符合语言习惯的表示方式。

Lambda表达式不仅可以用来创建委托实例，C#编译器也能够将他们转换成表达式树。

下面我们就先看看Lambda表达式。

作为委托的Lambda表达式

Lambda表达式可以看作是C# 2.0的匿名方法的进一步演变，所以匿名方法能做的几乎一切事情都可以用Lambda表达式来完成（注意，匿名方法可以忽略参数，Lambda表达式不具备这个特性）。

跟匿名方法类似，Lambda表达式有特殊的转换规则：表达式的类型本身并非委托类型，但它可以通过隐式或显式的发那个是转换为一个委托实例。匿名函数这个术语同时涵盖了匿名方法和Lambda表达式。

下面看看使用Lambda表达式获得字符串长度的例子，通过Lambda将得到更见简洁、易读的代码：

C#

static void Main(string[] args)

{

//使用C# 2.0中的匿名方法获取字符串长度

Func<string, int> strLength = delegate(string str) { return str.Length; };

Console.WriteLine(strLength("Hello World!"));

//使用Lambda表达式

//（显式类型参数列表）=> {语句}，lambda表达式最冗长版本

strLength = (string str) => { return str.Length; };

Console.WriteLine(strLength("Hello World!"));

//单一表达式作为主体

//（显式类型参数列表）=> 表达式

strLength = (string str) => str.Length;

Console.WriteLine(strLength("Hello World!"));

//隐式类型的参数列表

//（隐式类型参数列表）=> 表达式

strLength = (str) => str.Length;

Console.WriteLine(strLength("Hello World!"));

//单一参数的快捷语法

//参数名 => 表达式

strLength = str => str.Length;

Console.WriteLine(strLength("Hello World!"));

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

static void Main(string[] args)

{

//使用C# 2.0中的匿名方法获取字符串长度

Func<string, int> strLength = delegate(string str) { return str.Length; };

Console.WriteLine(strLength("Hello World!"));

//使用Lambda表达式

//（显式类型参数列表）=> {语句}，lambda表达式最冗长版本

strLength = (string str) => { return str.Length; };

Console.WriteLine(strLength("Hello World!"));

//单一表达式作为主体

//（显式类型参数列表）=> 表达式

strLength = (string str) => str.Length;

Console.WriteLine(strLength("Hello World!"));

//隐式类型的参数列表

//（隐式类型参数列表）=> 表达式

strLength = (str) => str.Length;

Console.WriteLine(strLength("Hello World!"));

//单一参数的快捷语法

//参数名 => 表达式

strLength = str => str.Length;

Console.WriteLine(strLength("Hello World!"));

}

“=>”是C# 3.0新增的，告诉编译器我们正在使用Lambda表达式。”=>”可以读作”goes to”，所以例子中的Lambda表达式可以读作”str goes to str.Length”。从例子中还可以看到，根据Lambda使用的特殊情况，我们可以进一步简化Lambda表达式。

Lambda表达式大多数时候都是和一个返回非void的委托类型配合使用（例如Func<TResult>）。在C# 1.0中，委托一般用于事件，很少会返回什么结果。在LINQ中，委托通常被视为数据管道的一部分，接受输入并返回结果，或者判断某项是否符合当前的筛选 器等等。

Lambda表达式本质

通过ILSpy查看上面的例子，可以发现Lambda表达式就是匿名方法，是编译器帮我们进行了转换工作，使我们可以直接使用Lambda表达式来进一步简化创建委托实例的代码。

在List<T>中使用Lambda表达式

前面简单的介绍了什么是Lambda表达式，下面通过一个例子进一步了解Lambda表达式。

在前面的文章中，我们也提到了一下List<T>的方法，例如FindAll方法，参数是Predicate<T>类型的 委托，返回结果是一个筛选后的新列表；Foreach方法获取一个Action<T>类型的委托，然后对每个元素设置行为。下面就看看在 List<T>中使用Lambda表达式：

C#

public class Book

{

public string Name { get; set; }

public int Year { get; set; }

}

class Program

{

static void Main(string[] args)

{

var books = new List<Book>

{

new Book{Name="C# learning guide",Year=2005},

new Book{Name="C# step by step",Year=2005},

new Book{Name="Java learning guide",Year=2004},

new Book{Name="Java step by step",Year=2004},

new Book{Name="Python learning guide",Year=2003},

new Book{Name="C# in depth",Year=2012},

new Book{Name="Java in depth",Year=2014},

new Book{Name="Python in depth",Year=2013},

};

//创建一个委托实例来表示一个通用的操作

Action<Book> printer = book => Console.WriteLine("Name = {0}, Year = {1}", book.Name, book.Year);

books.ForEach(printer);

//使用Lambda表达式对List<T>进行筛选

books.FindAll(book => book.Year > 2010).ForEach(printer);

books.FindAll(book => book.Name.Contains("C#")).ForEach(printer);

//使用Lambda表达式对List<T>进行排序

books.Sort((book1, book2) => book1.Name.CompareTo(book2.Name));

books.ForEach(printer);

Console.Read();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

public class Book

{

public string Name { get; set; }

public int Year { get; set; }

}

class Program

{

static void Main(string[] args)

{

var books = new List<Book>

{

new Book{Name="C# learning guide",Year=2005},

new Book{Name="C# step by step",Year=2005},

new Book{Name="Java learning guide",Year=2004},

new Book{Name="Java step by step",Year=2004},

new Book{Name="Python learning guide",Year=2003},

new Book{Name="C# in depth",Year=2012},

new Book{Name="Java in depth",Year=2014},

new Book{Name="Python in depth",Year=2013},

};

//创建一个委托实例来表示一个通用的操作

Action<Book> printer = book => Console.WriteLine("Name = {0}, Year = {1}", book.Name, book.Year);

books.ForEach(printer);

//使用Lambda表达式对List<T>进行筛选

books.FindAll(book => book.Year > 2010).ForEach(printer);

books.FindAll(book => book.Name.Contains("C#")).ForEach(printer);

//使用Lambda表达式对List<T>进行排序

books.Sort((book1, book2) => book1.Name.CompareTo(book2.Name));

books.ForEach(printer);

Console.Read();

}

}

从上面例子可以看到，当我们要经常使用一个操作的时候，我们最好创建一个委托实例，然后反复调用，而不是每次使用的时候都使用Lambda表达式（例如例子中的printer委托实例）。

相比C# 1.0中的委托或者C# 2.0的匿名函数，结合Lambda表达式，对List<T>中的数据操作变得简单，易读。

表达式树

表达式树也称表达式目录树，将代码以一种抽象的方式表示成一个对象树，树中每个节点本身都是一个表达式。表达式树不是可执行代码，它是一种数据结构。

下面我们看看怎么通过C#代码建立一个表达式树。

构建表达式树

System.Linq.Expressions命名空间中包含了代表表达式的各个类，所有类都从Expression派生，我们可以通过这些类中的静态方法来创建表达式类的实例。Expression类包括两个重要属性：

Type属性代表求值表达式的.NET类型，可以把它视为一个返回类型

NodeType属性返回所代表的表达式的类型

下面看一个构建表达式树的简单例子：

C#

Expression numA = Expression.Constant(6);

Console.WriteLine("NodeType: {0}, Type: {1}", numA.NodeType, numA.Type);

Expression numB = Expression.Constant(3);

Console.WriteLine("NodeType: {0}, Type: {1}", numB.NodeType, numB.Type);

BinaryExpression add = Expression.Add(numA, numB);

Console.WriteLine("NodeType: {0}, Type: {1}", add.NodeType, add.Type);

Console.WriteLine(add);

Console.Read();

1

2

3

4

5

6

7

8

9

10

Expression numA = Expression.Constant(6);

Console.WriteLine("NodeType: {0}, Type: {1}", numA.NodeType, numA.Type);

Expression numB = Expression.Constant(3);

Console.WriteLine("NodeType: {0}, Type: {1}", numB.NodeType, numB.Type);

BinaryExpression add = Expression.Add(numA, numB);

Console.WriteLine("NodeType: {0}, Type: {1}", add.NodeType, add.Type);

Console.WriteLine(add);

Console.Read();

代码的输出为：

通过例子可以看到，我们构建了一个(6+3)的表达式树，并且查看了各个节点的Type和NodeType属性。

Expression有很多派生类，有很多节点类型。例如，BinaryExpression就代表了具有两个操作树的任意操作。这正是NodeType属性重要的地方，它能区分由相同的类表示的不同种类的表达式。其他的节点类型就不介绍了，有兴趣可以参考MSDN。

对于上面的例子，可以用下图描述生成的表达式树，值得注意的是，”叶子”表达式在代码中是最先创建的，，表达式是自下而上构建的。表达式是不易变的，所有可以缓存和重用表达式。

将表达式编译成委托

LambdaExpression是从Expression派生的类型之一。泛型类型Expression<TDelegate>又是从LambdaExpress派生的。

Expression和Expression<TDelegate>的区别在于，泛型类以静态类型的方式标志了它是什么种类的表达式，也就是说，它确定了返回类型和参数。例如上面的加法例子，返回值是一个int类型，没有参数，所以我们可以使用签名Func<int>与之匹配，所以可以用Expression<Func<int>>以静态类型的方式来表示该表达式。

这样做的目的在于，LambdaExpression有一个Compile方法，该方法能创建一个恰当类型的委托。 Expression<TDelegate>也有一个同名方法，该方法可以返回TDelegate类型的委托。获得了委托之后，我们就可以使 用普通委托实例调用的方式来执行这个表达式。

接着上面加法的例子，我们把上面的加法表达式树转换成委托，然后执行委托：

C#

Func<int> addDelegate = Expression.Lambda<Func<int>>(add).Compile();

Console.WriteLine(addDelegate());

1

2

Func<int> addDelegate = Expression.Lambda<Func<int>>(add).Compile();

Console.WriteLine(addDelegate());

从这个例子中我们看到怎么构建一个表达式树，然后把这个对象树编译成真正的代码。在.NET 3.5中的表达式树只能是单一的表达式，不能表示完整的类、方法。这在.NET 4.0中得到了一定的改进，表达式树可以支持动态类型，我们可以创建块，为表达式赋值等等。

将Lambda表达式转换为表达式树

Lambda表达式不仅可以创建委托实例，C# 3.0对于将Lambda表达式转换成表达式树提供了内建的支持。我们可以通过编译器把Lambda表达式转换成一个表达式树，并创建一个Expression<TDelegate>的一个实例。

下面的例子中我们将一个Lambda表达式转换成一个表达式树，并通过代码查看表达式树的各个部分：

C#

static void Main(string[] args)

{

//将Lambda表达式转换为类型Expression<T>的表达式树

//expression不是可执行代码

Expression<Func<int, int, int>> expression = (a, b) => a + b;

Console.WriteLine(expression);

//获取Lambda表达式的主体

BinaryExpression body = (BinaryExpression)expression.Body;

Console.WriteLine(expression.Body);

//获取Lambda表达式的参数

Console.WriteLine(" param1: {0}, param2: {1}", expression.Parameters[0], expression.Parameters[1]);

ParameterExpression left = (ParameterExpression)body.Left;

ParameterExpression right = (ParameterExpression)body.Right;

Console.WriteLine(" left body of expression: {0}{4} NodeType: {1}{4} right body of expression: {2}{4} Type: {3}{4}", left.Name, body.NodeType, right.Name, body.Type, Environment.NewLine);

//将表达式树转换成委托并执行

Func<int, int, int> addDelegate = expression.Compile();

Console.WriteLine(addDelegate(10, 16));

Console.Read();

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

static void Main(string[] args)

{

//将Lambda表达式转换为类型Expression<T>的表达式树

//expression不是可执行代码

Expression<Func<int, int, int>> expression = (a, b) => a + b;

Console.WriteLine(expression);

//获取Lambda表达式的主体

BinaryExpression body = (BinaryExpression)expression.Body;

Console.WriteLine(expression.Body);

//获取Lambda表达式的参数

Console.WriteLine(" param1: {0}, param2: {1}", expression.Parameters[0], expression.Parameters[1]);

ParameterExpression left = (ParameterExpression)body.Left;

ParameterExpression right = (ParameterExpression)body.Right;

Console.WriteLine(" left body of expression: {0}{4} NodeType: {1}{4} right body of expression: {2}{4} Type: {3}{4}", left.Name, body.NodeType, right.Name, body.Type, Environment.NewLine);

//将表达式树转换成委托并执行

Func<int, int, int> addDelegate = expression.Compile();

Console.WriteLine(addDelegate(10, 16));

Console.Read();

}

代码的输出为：

表达式树的用途

前面看到，通过Expression的派生类中的各种节点类型，我们可以构建表达式树；然后可以把表达式树转换成相应的委托类型实例，最后执行委托实例的代码。但是，我们不会绕这么大的弯子来执行委托实例的代码。

表达式树主要在LINQ to SQL中使用，我们需要将LINQ to SQL查询表达式（返回IQueryable类型）转换成表达式树。之所以需要转换是因为LINQ to SQL查询表达式不是在C#代码中执行的，LINQ to SQL查询表达式被转换成SQL，通过网络发送，最后在数据库服务器上执行。

这里只做个简单的介绍，后续会介绍LINQ to SQL相关的内容。

编译器对Lambda表达式的处理

前面我们了解到，Lambda可以用来创建委托实例，也可以用来生成表达式树，这些都是编译器帮我们完成的。

编译器如何决定生成可执行的IL还是一个表达式树：

当Lambda表达式赋予一个委托类型的变量时，编译器生成与匿名方法同样的IL（可执行的委托实例）

当Lambda表达式赋予一个Expression类型的变量时，编译器就将它转换成一个表达式树

下图展示了LINQ to Object和LINQ to SQL中Lambda表达式的不同处理方式：

总结

本文中介绍了Lambda表达式，在匿名方法的基础上进一步简化了委托实例的创建，编写更加简洁、易读的代码。匿名函数不等于匿名方法，匿名函数包含了匿名方法和lambda表达式这两种概念

Lambda不仅可以创建委托实例，还可以由编译器转换成表达式树，使代码可以在程序之外执行（参考LINQ to SQL）。