Linq学习总结

Lambda表达式

Lambda表达式的演化

要了解Lambda表达式，我们首先应从**委托**说起。.NET中的委托实际上就是C语言中的函数指针，函数通过地址进行引用，只不过.NET中它更加好看了而已。

```
delegate void FunctionPoint(string str);
static void printHello(string name)
{
      Console.WriteLine("Hello {0}", name);
}
static void Main(string[] args)
{
      FunctionPoint fp = printHello;
      fp("heqichang");
}
 /*Ouput
 * Hello heqichang
 */
```

然后，委托在.NET2.0中又被精简成了**匿名委托**

```
delegate void FunctionPoint(string str);
 static void Main(string[] args)
{
    FunctionPoint fp = delegate(string name)
    {
        Console.WriteLine("Hello {0}",name);
    };
    fp("heqichang");
}
/*Ouput
 * Hello heqichang
 */
```

匿名委托省略了函数名、返回类型以及参数类型，变得更加轻量

```
delegate void FunctionPoint(string str);
static void Main(string[] args)
{
    FunctionPoint fp = s => Console.WriteLine("Hello {0}",s);
    fp("heqichang");
}
/*Ouput
 * Hello heqichang
 */
```

现在我们看到的表达式就是通过委托一步一步简化而来的。我们的Lambda表达式就是一个简洁的委托。左侧（相对于=>）代表函数的参数，右侧就是函数体。

在System命名空间中，微软已经为我们预定义了几个**泛型委托Action**、**Func**、**Predicate**。Action用于在泛型参数上执行一个操作；Func用于在参数上执行一个操作并返回一个值；Predicate<T>用于定义一组条件并确定参数是否符合这些条件。

表达式树

Lambda表达式还有个重要的用途就是用来**构建表达式树**：

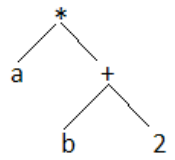
```
static void Main(string[] args)
{
    Expression<Func<int, int, int>> exp = (a, b) => a * (b + 2);

    ParameterExpression param1 = (ParameterExpression)exp.Parameters[0];
    ParameterExpression param2 = (ParameterExpression)exp.Parameters[1];
    BinaryExpression operation = (BinaryExpression)exp.Body;
    ParameterExpression left = (ParameterExpression)operation.Left;
    BinaryExpression operation2 = (BinaryExpression)operation.Right;
    ParameterExpression left2 = (ParameterExpression)operation2.Left;
    ConstantExpression right2 = (ConstantExpression)operation2.Right;

    Console.WriteLine("Decomposed expression: ({0},{1}) => {2} {3} ({4} {5} {6})",
                      param1.Name, param2.Name, left.Name, operation.NodeType, left2.Name,
                      operation2.NodeType, right2.Value);

    Func<int, int, int> func = exp.Compile();
    Console.WriteLine(func(2,2));
    /*Ouput
     * 8
     */
}
```

我们上面的lambda表达式构建了这么一个表达式树：

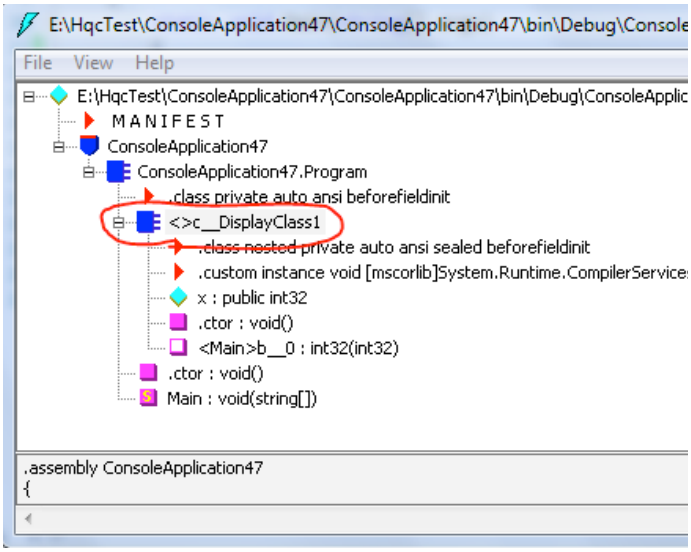


闭包

如果将一个变量声明在一个函数内部，该变量就只会在该函数的栈内存中。当函数返回，这个本地变量也同时从栈内存中被清除了。当你在Lambda表达式中使用本地变量时，该变量就会在函数的栈空间清理时被移除。为了防止这样的事发生，当一个依赖于本地变量的Lambda表达式需要从函数中返回时，编译器就会创建一个闭包（Closure，也就是一个包装器类）。

```
static void Main(string[] args)
{
    int x = 1;
    Func<int, int> add = y => x + y;
    Console.WriteLine(add(3));
    /*Output
     * 4
     */
}
```

我们通过ILDasm可以看到，编译器帮我们自动创建了一个类，用于保存本地变量，以扩展它们的生命周期


E:\HqcTest\ConsoleApplication47\ConsoleApplication47\bin\Debug\Console
File View Help
E:\HqcTest\ConsoleApplication47\ConsoleApplication47\bin\Debug\ConsoleApplic
MANIFEST
ConsoleApplication47
ConsoleApplication47.Program
.class private auto ansi beforefieldinit
<>c__DisplayClass1
.class nested private auto ansi sealed beforefieldinit
.custom instance void [mscorlib]System.Runtime.CompilerService
x : public int32
.ctor : void()
<Main>b__0 : int32(int32)
.ctor : void()
Main : void(string[])
.assembly ConsoleApplication47
{