Lambda表达式的前世今生 - 文章 - 伯乐在线

Lambda 表达式

早在 C# 1.0 时，C#中就引入了委托（delegate）类型的概念。通过使用这个类型，我们可以将函数作为参数进行传递。在某种意义上，委托可理解为一种托管的强类型的函数指针。

通常情况下，使用委托来传递函数需要一定的步骤：

1. 定义一个委托，包含指定的参数类型和返回值类型。
2. 在需要接收函数参数的方法中，使用该委托类型定义方法的参数签名。
3. 为指定的被传递的函数创建一个委托实例。

可能这听起来有些复杂，不过本质上说确实是这样。上面的第 3 步通常不是必须的，C# 编译器能够完成这个步骤，但步骤 1 和 2 仍然是必须的。

幸运的是，在 C# 2.0 中引入了泛型。现在我们能够编写泛型类、泛型方法和最重要的：泛型委托。尽管如此，直到 .NET 3.5，微软才意识到实际上仅通过两种泛型委托就可以满足 99% 的需求：

- Action ：无输入参数，无返回值
- Action ：支持1-16个输入参数，无返回值
- Func ：支持1-16个输入参数，有返回值

Action 委托返回 void 类型，Func 委托返回指定类型的值。通过使用这两种委托，在绝大多数情况下，上述的步骤 1 可以省略了。但是步骤 2 仍然是必需的，但仅是需要使用 Action 和 Func。

那么，如果我只是想执行一些代码该怎么办？在 C# 2.0 中提供了一种方式，创建匿名函数。但可惜的是，这种语法并没有流行起来。下面是一个简单的匿名函数的示例：

C#

```
Funcdouble, double> square = delegate(double x)
{
    return x * x;
};
```

为了改进这些语法，在 .NET 3.5 框架和 C# 3.0 中引入了Lambda 表达式。

首先我们先了解下 Lambda 表达式名字的由来。实际上这个名字来自微积分数学中的 λ，其涵义是声明为了表达一个函数具体需要什么。更确切的说，它描述了一个数学逻辑系统，通过变量结合和替换来表达计算。所以，基本上我们有 0-n 个输入参数和一个返回值。而在编程语言中，我们也提供了无返回值

的 void 支持。

让我们来看一些 Lambda 表达式的示例：

C#

```
        // The compiler cannot resolve this, which makes the usage of var impossible!
        // Therefore we need to specify the type.
        Action dummyLambda = () =>
        {
            Console.WriteLine("Hello World from a Lambda expression!");
        };
        // Can be used as with double y = square(25);
        Funcdouble, double> square = x => x * x;
    // Can be used as with double z = product(9, 5);
    Funcdouble, double, double> product = (x, y) => x * y;
    // Can be used as with printProduct(9, 5);
    Actiondouble, double> printProduct = (x, y) => { Console.WriteLine(x * y); };
    // Can be used as with
    // var sum = dotProduct(new double[] { 1, 2, 3 }, new double[] { 4, 5, 6 });
    Funcdouble[], double[], double> dotProduct = (x, y) =>
    {
        var dim = Math.Min(x.Length, y.Length);
        var sum = 0.0;
        for (var i = 0; i != dim; i++)
            sum += x[i] + y[i];
        return sum;
    };
    // Can be used as with var result = matrixVectorProductAsync(...);
    Funcdouble[,], double[], Taskdouble[]>> matrixVectorProductAsync =
        async (x, y) =>
        {
            var sum = 0.0;
            /* do some stuff using await ... */
            return sum;
        };
```

从这些语句中我们可以直接地了解到：

- 如果仅有一个入参，则可省略圆括号。
- 如果仅有一行语句，并且在该语句中返回，则可省略大括号，并且也可以省略 return 关键字。
- 通过使用 async 关键字，可以将 Lambda 表达式声明为异步执行。
- 大多数情况下，var 声明可能无法使用，仅在一些特殊的情况下可以使用。

在使用 var 时，如果编译器通过参数类型和返回值类型推断无法得出委托类型，将会抛出 “Cannot

assign lambda expression to an implicitly-typed local variable.” 的错误提示。来看下如下这些示例：

```
var square = (double x) => x * x;
var stringLengthSquare = (string s) => s.Length * s.Length;

var squareAndOutput = (decimal x, string s) =>
{
  var sqz = x * x;
  Console.WriteLine("Information by {0}: the square of {1} is {2}.", s, x, sqz);
};
```

Cannot assign lambda expression to an implicitly-typed local variable

现在我们已经了解了大部分基础知识，但一些 Lambda 表达式特别酷的部分还没提及。

我们来看下这段代码：

C#

```
var a = 5;
Funcint, int> multiplyWith = x => x * a;
var result1 = multiplyWith(10); // 50
a = 10;
var result2 = multiplyWith(10); // 100
```

可以看到，在 Lambda 表达式中可以使用外围的变量，也就是闭包。

C#

```
static void DoSomeStuff()
{
    var coeff = 10;
    Funcint, int> compute = x => coeff * x;
    Action modifier = () =>
    {
        coeff = 5;
    };
    var result1 = DoMoreStuff(compute); // 50
    ModifyStuff(modifier);
    var result2 = DoMoreStuff(compute); // 25
}
static int DoMoreStuff(Funcint, int> computer)
{
    return computer(5);
}
static void ModifyStuff(Action modifier)
{
    modifier();
}
```

这里发生了什么呢？首先我们创建了一个局部变量和两个 Lambda 表达式。第一个 Lambda 表达式展示了其可以在其他作用域中访问该局部变量，实际上这已经展现了强大的能力了。这意味着我们可以保护一个变量，但仍然可以在其他方法中访问它，而不用关心那个方法是定义在当前类或者其他类中。

第二个 Lambda 表达式展示了在 Lambda 表达式中能够修改外围变量的能力。这就意味着通过在函数间传递 Lambda 表达式，我们能够在其他方法中修改其他作用域中的局部变量。因此，我认为闭包是一种特别强大的功能，但有时也可能引入一些非期望的结果。

C#

```
var buttons = new Button[10];
for (var i = 0; i )
{
    var button = new Button();
    button.Text = (i + 1) + ". Button - Click for Index!";
    button.OnClick += (s, e) => { Messagebox.Show(i.ToString()); };
    buttons[i] = button;
}
//What happens if we click ANY button?!
```

这个诡异的问题的结果是什么呢？是 Button 0 显示 0, Button 1 显示 1 吗？答案是：所有的 Button 都显示 10！

因为随着 for 循环的遍历，局部变量 i 的值已经被更改为 buttons 的长度 10。一个简单的解决办法类似于：

```
1
2
3
4
5
```

```
var button = new Button();
var index = i;
button.Text = (i + 1) + ". Button - Click for Index!";
button.OnClick += (s, e) => { Messagebox.Show(index.ToString()); };
buttons[i] = button;
```

通过定义变量 index 来拷贝变量 i 中的值。

注：如果你使用 Visual Studio 2012 以上的版本进行测试，因为使用的编译器与 Visual Studio 2010 的不同，此处测试的结果可能不同。可参考：Visual C# Breaking Changes in Visual Studio 2012

表达式树

在使用 Lambda 表达式时，一个重要的问题是目标方法是怎么知道如下这些信息的：

1. 我们传递的变量的名字是什么？

2. 我们使用的表达式体的结构是什么？
3. 在表达式体内我们用了哪些类型？

现在，表达式树帮我们解决了问题。它允许我们深究具体编译器是如何生成的表达式。此外，我们也可以执行给定的函数，就像使用 Func 和 Action 委托一样。其也允许我们在运行时解析 Lambda 表达式。

我们来看一个示例，描述如何使用 Expression 类型：

C#

```
Expressionint>> expr = model => model.MyProperty;
var member = expr.Body as MemberExpression;
var propertyName = memberExpression.Member.Name; //only execute if member != null
```

上面是关于 Expression 用法的一个最简单的示例。其中的原理非常直接：通过形成一个 Expression 类型的对象，编译器会根据表达式树的解析生成元数据信息。解析树中包含了所有相关的信息，例如参数和方法体等。

方法体包含了整个解析树。通过它我们可以访问操作符、操作对象以及完整的语句，最重要的是能访问返回值的名称和类型。当然，返回变量的名称可能为 null。尽管如此，大多数情况下我们仍然对表达式的内容很感兴趣。对于开发人员的益处在于，我们不再会拼错属性的名称，因为每个拼写错误都会导致编译错误。

如果程序员只是想知道调用属性的名称，有一个更简单优雅的办法。通过使用特殊的参数属性 CallerMemberName 可以获取到被调用方法或属性的名称。编译器会自动记录这些名称。所以，如果我们仅是需要获知这些名称，而无需更多的类型信息，则我们可以参考如下的代码写法：

C#

```
string WhatsMyName([CallerMemberName] string callingName = null)
{
        return callingName;
}
```

Lambda 表达式的性能

有一个大问题是：Lambda 表达式到底有多快？当然，我们期待其应该与常规的函数一样快，因为 Lambda 表达式也同样是由编译器生成的。在下一节中，我们会看到为 Lambda 表达式生成的 MSIL 与常规的函数并没有太大的不同。

一个非常有趣的讨论是关于在 Lambda 表达式中的闭包是否要比使用全局变量更快，而其中最有趣的地方就是是否当可用的变量都在本地作用域时是否会有性能影响。

让我们来看一些代码，用于衡量各种性能基准。通过这 4 种不同的基准测试，我们应该有足够的证据来说明常规函数与 Lambda 表达式之间的不同了。

C#

```
class StandardBenchmark : Benchmark
{
    static double[] A;
    static double[] B;
    public static void Test()
    {
        var me = new StandardBenchmark();
     Init();
     for (var i = 0; i 10; i++)
     {
         var lambda = LambdaBenchmark();
         var normal = NormalBenchmark();
         me.lambdaResults.Add(lambda);
         me.normalResults.Add(normal);
     }
     me.PrintTable();
    }
 static void Init()
 {
     var r = new Random();
     A = new double[LENGTH];
     B = new double[LENGTH];
     for (var i = 0; i )
     {
         A[i] = r.NextDouble();
         B[i] = r.NextDouble();
     }
 }
 static long LambdaBenchmark()
 {
     Funcdouble> Perform = () =>
     {
         var sum = 0.0;
         for (var i = 0; i )
             sum += A[i] * B[i];
         return sum;
     };
     var iterations = new double[100];
     var timing = new Stopwatch();
     timing.Start();
     for (var j = 0; j )
```

```
                iterations[j] = Perform();
            timing.Stop();
            Console.WriteLine("Time for Lambda-Benchmark: t {0}ms",
                timing.ElapsedMilliseconds);
            return timing.ElapsedMilliseconds;
        }
        static long NormalBenchmark()
        {
            var iterations = new double[100];
            var timing = new Stopwatch();
            timing.Start();
            for (var j = 0; j )
                iterations[j] = NormalPerform();
            timing.Stop();
            Console.WriteLine("Time for Normal-Benchmark: t {0}ms",
                timing.ElapsedMilliseconds);
            return timing.ElapsedMilliseconds;
        }
        static double NormalPerform()
        {
            var sum = 0.0;
            for (var i = 0; i )
                sum += A[i] * B[i];
            return sum;
        }
    }
```

当然，利用 Lambda 表达式，我们可以把上面的代码写的更优雅一些，这么写的原因是防止干扰最终的结果。所以我们仅提供了 3 个必要的方法，其中一个负责执行 Lambda 测试，一个负责常规函数测试，第三个方法则是在常规函数。而缺少的第四个方法就是我们的 Lambda 表达式，其已经在第一个方法中内嵌了。使用的计算方法并不重要，我们使用了随机数，进而避免了编译器的优化。最后，我们最感兴趣的就是常规函数与 Lambda 表达式的不同。

在运行这些测试后，我们会发现，在通常情况下 Lambda 表达式不会表现的比常规函数更差。而其中的一个很奇怪的结果就是，Lambda 表达式实际上在某些情况下表现的要比常规方法还要好些。当然，如果是在使用闭包的条件下，结果就不一样了。这个结果告诉我们，使用 Lambda 表达式无需再犹豫。但是我们仍然需要仔细的考虑当我们使用闭包时所丢失的性能。在这种情景下，我们通常会丢失一点性能，但或许仍然还能接受。关于性能丢失的原因将在下一节中揭开。

下面的表格中显示了基准测试的结果：

- 无入参无闭包比较

```
                Lambda [ms]     Normal [ms]
------------------------------
Best            47              47
Worst           47              47
Average         47              47
Geo. Mean       47              47
Variance        0               0
```

- 含入参比较

```
                Lambda [ms]     Normal [ms]
------------------------------
Best            47              47
Worst           47              48
Average         47              47
Geo. Mean       47              47.0990552591869
Variance        0               0.0900000000001455
```

- 含闭包比较

```
                Lambda [ms]     Normal [ms]
------------------------------
Best            51              47
Worst           52              50
Average         51              48
Geo. Mean       51.5976683390374        47.9938332415664
Variance        0.240000000000146       0.6
```

- 含入参含闭包比较

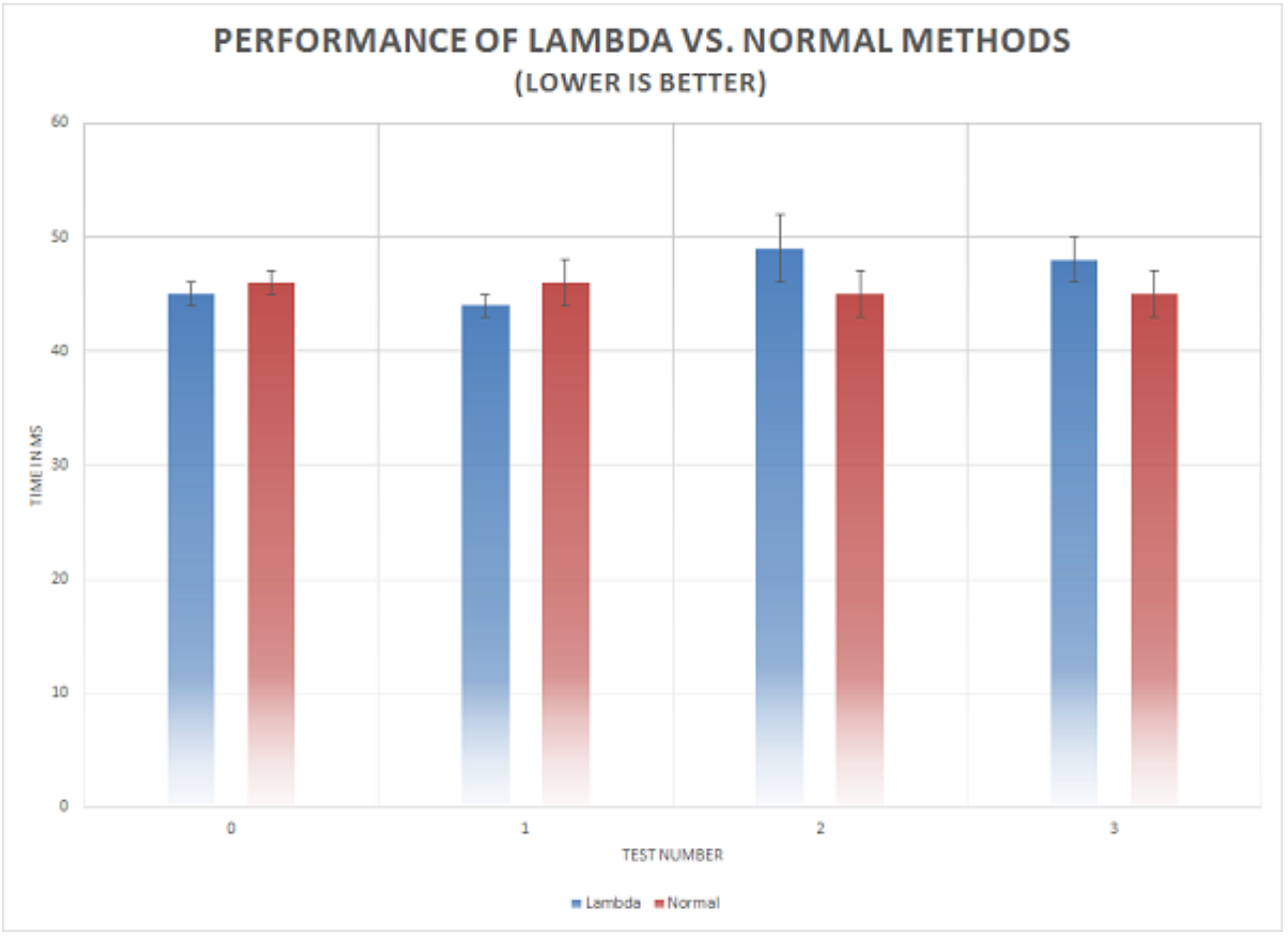
```
                Lambda [ms]     Normal [ms]
------------------------------
Best            51              47
Worst           52              48
Average         51              47
Geo. Mean       51.3976713555013        47.5974718403308
Variance        0.240000000000146       0.240000000000146
```

Test	Lambda [ms]	Normal [ms]
0	45+-1	46+-1
1	44+-1	46+-2
2	49+-3	45+-2
3	48+-2	45+-2

注：测试结果根据机器硬件配置有所不同

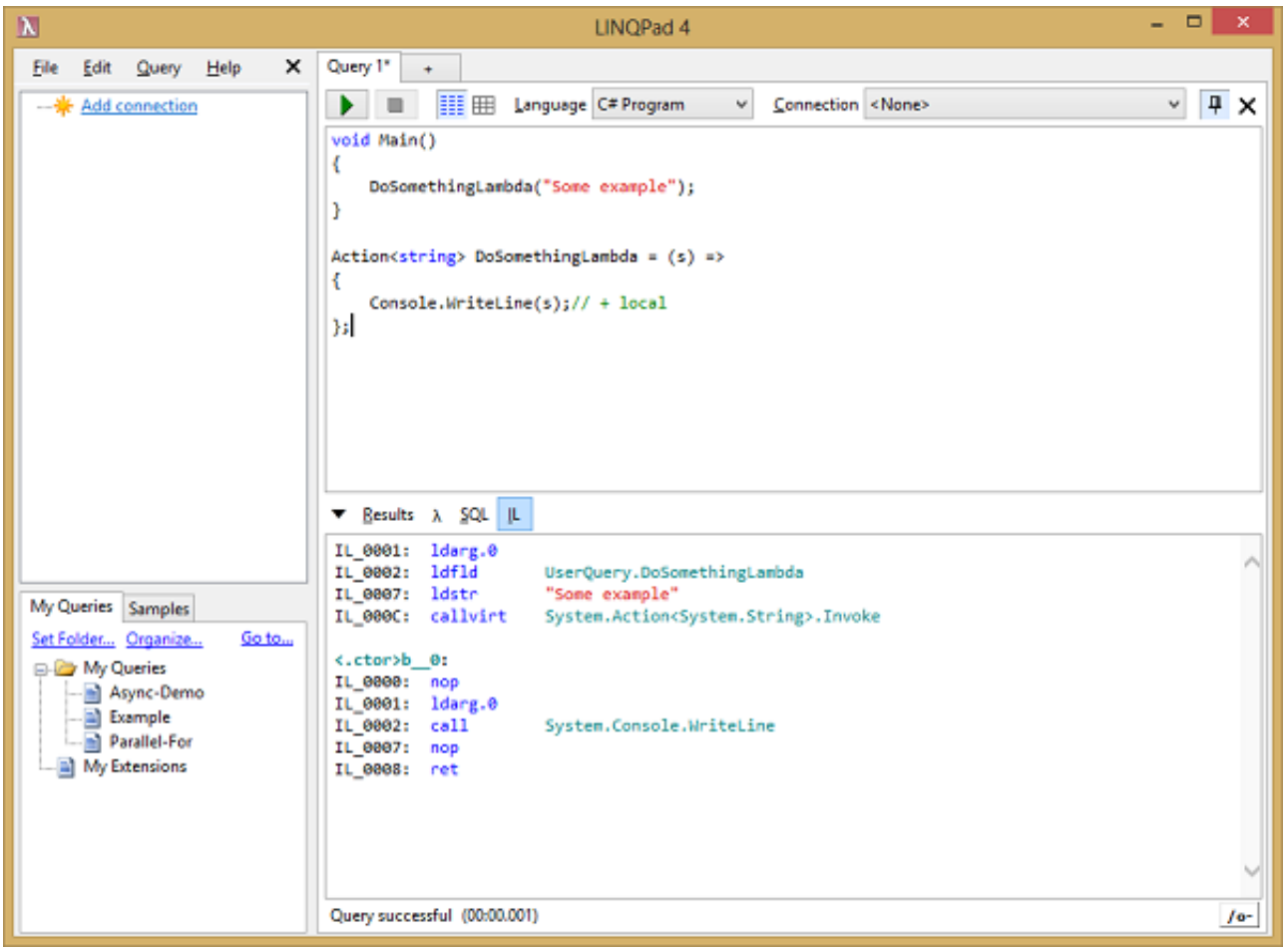
下面的图表中同样展现了测试结果。我们可以看到，常规函数与 Lambda 表达式会有相同的限制。使用

Lambda 表达式并没有显著的性能损失。



MSIL揭秘Lambda表达式

使用著名的工具 LINQPad 我们可以查看 MSIL。



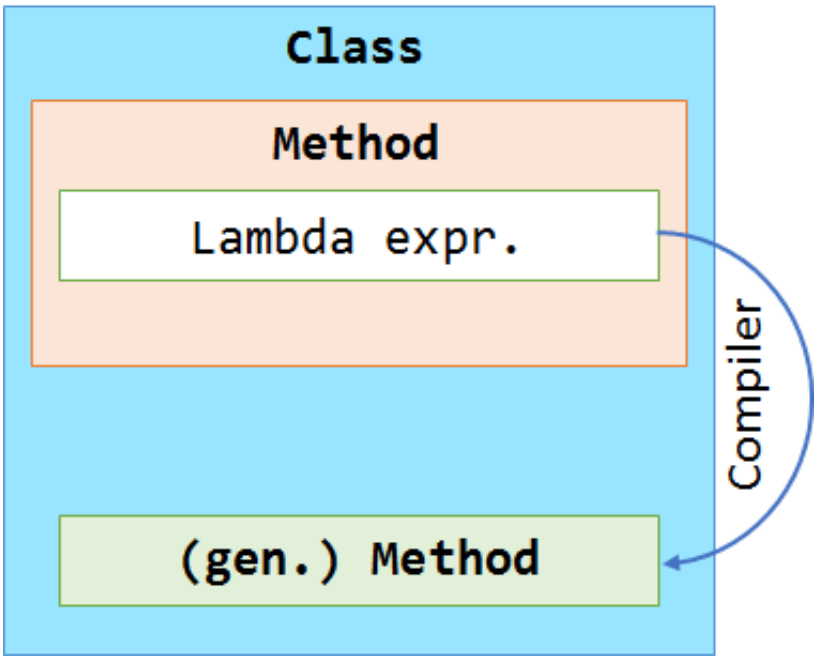
我们来看下第一个示例：

```
1
2
3
4
5
    IL_0001:    ldarg.0
    IL_0002:    ldfld             UserQuery.DoSomethingLambda
    IL_0007:    ldstr             "some example"
    IL_000C:    callvirt        System.Action.Invoke
    IL_0011:    nop
    IL_0012:    ldarg.0
    IL_0013:    ldstr             "some example"
    IL_0018:    call                UserQuery.DoSomethingNormal
DoSomethingNormal:
IL_0000:    nop
IL_0001:    ldarg.1
IL_0002:    call              System.Console.WriteLine
IL_0007:    nop
IL_0008:    ret
b__0:
IL_0000:    nop
```

```
IL_0001:    ldarg.0
IL_0002:    call                System.Console.WriteLine
IL_0007:    nop
IL_0008:    ret
```

此处最大的不同就是函数的命名和用法，而不是声明方式，实际上声明方式是相同的。编译器会在当前类中创建一个新的方法，然后推断该方法的用法。这没什么特别的，只是使用 Lambda 表达式方便了许多。从 MSIL 的角度来看，我们做了相同的事，也就是在当前的对象上调用了一个方法。

我们可以将这些分析放到一张图中，来展现编译器所做的更改。在下面这张图中我们可以看到编译器将 Lambda 表达式移到了一个单独的方法中。



在第二个示例中，我们将展现 Lambda 表达式真正神奇的地方。在这个例子中，我们使用了一个常规的方法来访问全局变量，然后用一个 Lambda 表达式来捕获局部变量。代码如下：

C#

```
    IL_0000:    newobj            UserQuery+c__DisplayClass1..ctor
    IL_0005:    stloc.1           // CS$8__locals2
    IL_0006:    nop
    IL_0007:    ldloc.1           // CS$8__locals2
    IL_0008:    ldc.i4.5
    IL_0009:    stfld             UserQuery+c__DisplayClass1.local
    IL_000E:    ldloc.1           // CS$8__locals2
    IL_000F:    ldftn             UserQuery+c__DisplayClass1.b__0
    IL_0015:    newobj            System.Action..ctor
    IL_001A:    stloc.0           // DoSomethingLambda
    IL_001B:    ldarg.0
    IL_001C:    ldloc.1           // CS$8__locals2
    IL_001D:    ldfld             UserQuery+c__DisplayClass1.local
    IL_0022:    stfld             UserQuery.global
```

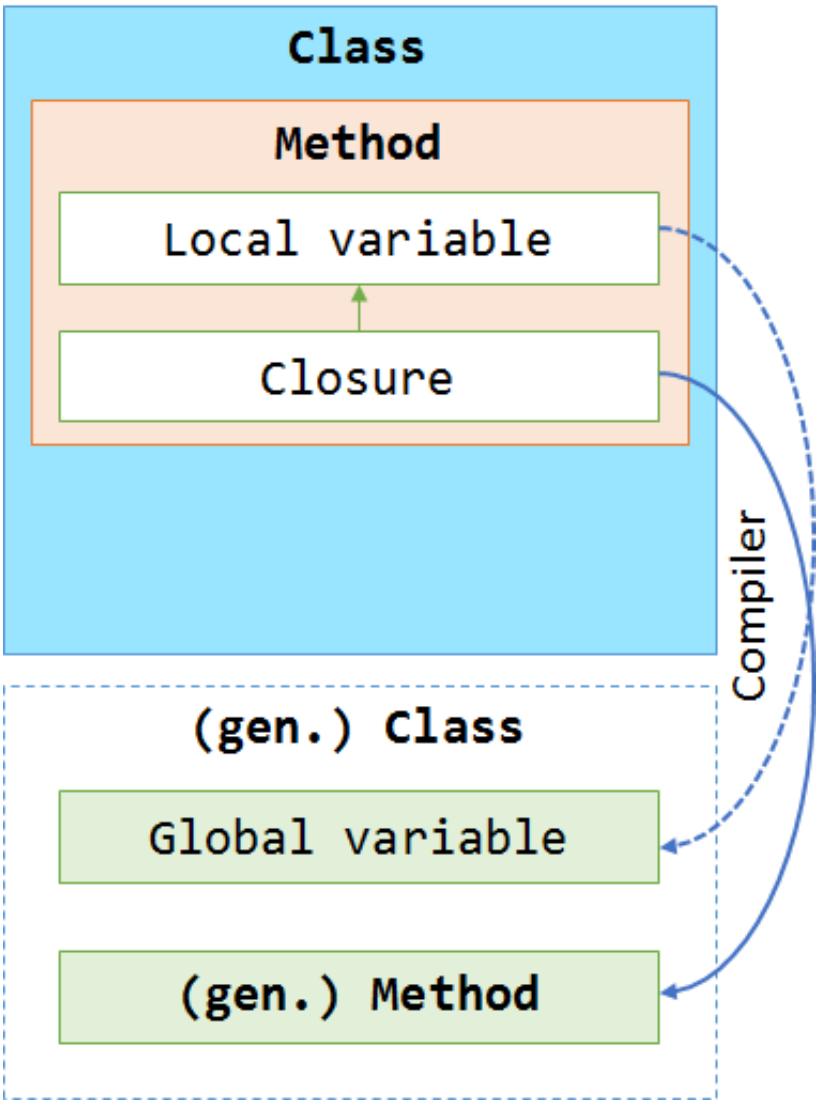
```
IL_0027:    ldloc.0          // DoSomethingLambda
IL_0028:    ldstr              "Test 1"
IL_002D:    callvirt         System.Action.Invoke
IL_0032:    nop
IL_0033:    ldarg.0
IL_0034:    ldstr              "Test 2"
IL_0039:    call               UserQuery.DoSomethingNormal
IL_003E:    nop
DoSomethingNormal:
IL_0000:    nop
IL_0001:    ldarg.1
IL_0002:    ldarg.0
IL_0003:    ldfld              UserQuery.global
IL_0008:    box                System.Int32
IL_000D:    call               System.String.Concat
IL_0012:    call               System.Console.WriteLine
IL_0017:    nop
IL_0018:    ret
c__DisplayClass1.b__0:
IL_0000:    nop
IL_0001:    ldarg.1
IL_0002:    ldarg.0
IL_0003:    ldfld              UserQuery+c__DisplayClass1.local
IL_0008:    box                System.Int32
IL_000D:    call               System.String.Concat
IL_0012:    call               System.Console.WriteLine
IL_0017:    nop
IL_0018:    ret
c__DisplayClass1..ctor:
IL_0000:    ldarg.0
IL_0001:    call               System.Object..ctor
IL_0006:    ret
```

还是一样，两个函数从调用语句上看是相同的，还是应用了与之前相同的机制。也就是说，编译器为该函数生成了一个名字，并把它替换到代码中。而此处最大的区别在于，编译器同时生成了一个类，而编译器生成的函数就被放到了这个类中。那么，创建这个类的目的是什么呢？它使变量具有了全局作用域范围，而此之前其已被用于捕获变量。通过这种方式，Lambda 表达式有能力访问局部作用域的变量（因为从 MSIL 的观点来看，其仅是类实例中的一个全局变量而已）。

然后，通过这个新生成的类的实例，所有的变量都从这个实例分配和读取。这解决了变量间存在引用的问题（会对类添加一个额外的引用 – 确实是这样）。编译器已经足够的聪明，可以将那些被捕获变量放到这个类中。所以，我们可能会期待使用 Lambda 表达式并不会存在性能问题。然而，这里我们必须提出一个警告，就是这种行为可能会引起内存泄漏，因为对象仍然被 Lambda 表达式引用着。只要这个

函数还在，其作用范围仍然有效（之前我们已经了解了这些，但现在我们知道了原因）。

像之前一样，我们把这些分析放入一张图中。从图中我们可以看到，闭包并不是仅有的被移动的方法，被捕获变量也被移动了。所有被移动的对象都会被放入一个编译器生成的类中。最后，我们从一个未知的类实例化了一个对象。



文章内容翻译并改编自 Way to Lambda ，章节和代码有很大的改动，未包含全部内容。

1 赞 1 收藏 评论

- 头条 - 分享和发现有价值的内容与观点
- 相亲 - 为IT单身男女服务的征婚传播平台
- 资源 - 优秀的工具资源导航
- 翻译 - 翻译传播优秀的外文文章
- 文章 - 国内外的精选文章
- 设计 - UI,网页，交互和用户体验
- iOS - 专注iOS技术分享
- 安卓 - 专注Android技术分享
- 前端 - JavaScript, HTML5, CSS
- Java - 专注Java技术分享
- Python - 专注Python技术分享