C#锐利体验2.0：

匿名方法、迭代器

李建忠
www.lijianzhong.com

上海视成科技 高级讲师

Agenda

- 使用匿名方法
- 匿名方法机制
- 使用迭代器
- 迭代器机制
- 讲座总结
- Q&A

匿名方法的由来

- 没有匿名方法的时候（C# 1.0）：

```
addButton.Click += new EventHandler (AddClick) ;

void AddClick(object sender, EventArgs e) {
      listBox.Items.Add(textBox.Text);
}
```

- 有了匿名方法之后（C# 2.0）：

```
addButton.Click += delegate {
            listBox.Items.Add(textBox.Text);
      };
```

匿名方法简介

- 匿名方法允许我们以一种“内联”的方式来编写方法代码，将代码直接与委托实例相关联，从而使得委托实例化的工作更加直观和方便。

- 匿名方法的几个相关问题：
  - 参数列表
  - 返回值
  - 外部变量

匿名方法的参数

- 匿名方法可以在delegate关键字后跟一个参数列表（可以不指定），后面的代码块则可以访问这些参数：

```
addButton.Click +=
        delegate(object sender, EventArgs e) {
                MessageBox.Show(((Button)sender).Text);
        };
```

- 注意“不指定参数列表”与“参数列表为空”的区别

```
addButton.Click +=  delegate {…}    // 正确！
addButton.Click +=  delegate() {…}    // 错误！
```

匿名方法的返回值

- 如果委托类型的返回值类型为void，匿名方法里便不能返回任何值；
- 如果委托类型的返回值类型不为void，匿名方法里返回的值必须和委托类型的返回值兼容：

```
delegate void MyDelegate();
MyDelegate d = delegate{
                   ……
                   return;
               };
```

```
delegate int MyDelegate();
MyDelegate d = delegate{
                   ……
                   return 100;
               };
```

匿名方法的外部变量

- 一些局部变量和参数有可能被匿名方法所使用，它们被称为“匿名方法的外部变量”。

- 外部变量的生存期会由于“匿名方法的捕获效益”而延长——一直延长到委托实例不被引用为止。

```
static void Foo(double factor) {
        Function f=delegate(int x) {
                factor +=0. 2;  // factor为外部变量
                return x * factor;
        };
        Invoke(f);  // factor的生存期被延长
}
```

委托类型的推断

- C# 2.0 允许我们在进行委托实例化时，省略掉委托类型，而直接采用方法名，C#编译器会做合理的推断。

- C# 1.0中的做法：

```
addButton.Click += new EventHandler(AddClick);
Apply(a, new Function(Math.Sin));
```

- C# 2.0中的做法：

```
addButton.Click += AddClick;
Apply(a, Math.Sin);
```

Agenda

- 使用匿名方法
- 匿名方法机制
- 使用迭代器
- 迭代器机制
- 讲座总结
- Q&A

匿名方法机制

- C# 2.0中的匿名方法仅仅是通过编译器的一层额外处理，来简化委托实例化的工作。它与C# 1.0的代码不存在根本性的差别。

- 通过ILDasm.exe反汇编工具，我们可以获得对匿名方法的深入了解：
  - 静态方法中的匿名方法
  - 实例方法中的匿名方法
  - 匿名方法中的外部变量

静态方法中的匿名方法

```
public delegate void D();
static void F() {
        D d = delegate { Console.WriteLine("test"); };
    }
}
```

上面的代码被编译器转换为：

```
static void F() {
    D d = new D(__Method1);
}

static void __Method1() {
    Console.WriteLine("test");
}
```

实例方法中的匿名方法

```
class Test {
    int x;
    void F() {
        D d = delegate { Console.WriteLine(this.x); };
    }
}
```

上面的代码被编译器转换为：

```
void F() {
    D d = new D(__Method1);
}

void __Method1() {
    Console.WriteLine(this.x);
}
```

匿名方法中的外部变量

```
void F() {
        int y = 123;
        D d = delegate { Console.WriteLine(y); };
    }
```

上面的代码被编译器转换为：

```
class __Temp
{
    public int y;
    public void __Method1() {
        Console.WriteLine(y);
    }
}
```

```
void F() {
    __Temp t = new __Temp();
    t.y = 123;
    D d = new
      D(t.__Method1);
}
```

Agenda

- 使用匿名方法
- 匿名方法机制
- 使用迭代器
- 迭代器机制
- 讲座总结
- Q&A

C# 1.0中的foreach

- 没有迭代器的时候，创建一个可用于foreach的集合（C# 1.0）：

```
public class MyCollection : IEnumerable {
    public MyEnumerator GetEnumerator() {
        return new MyEnumerator(this);
    }
    public class MyEnumerator : IEnumerator {
        public void Reset(){ …   }
        public bool MoveNext(){    …   }
        public int Current{   get{    …   }          }
        object IEnumerator.Current { get{  …  } }
    }
}
```

C# 1.0中的foreach

- 对集合使用foreach语句：

  foreach (int i in col) { ….. }

- 相当于：

```
IEnumerator etor =
    ((IEnumerable)col).GetEnumerator();
try {
   while (etor.MoveNext()) {
      ElementType elem = (ElementType)etor.Current;
      …….;
   }
}
finally {   ((IDisposable ) enumerator). Dispose();   }
```

C# 2.0中的迭代器

- 使用迭代器创建用于foreach的集合（C# 2.0）：

```
public class Stack<T>: IEnumerable<T>
{
    T[] items;
    int count;
    public void Push(T data) {...}
    public T Pop() {...}
    public IEnumerator<T> GetEnumerator() {
        for (int i = count – 1; i >= 0; --i) {
            yield return items[i];
        }
    }
}
```

C# 2.0中的迭代器

- 使用foreach语句：

```
Stack<int> stack = new Stack<int>();
foreach (int i in stack) {  ……. }
```

- 使用迭代器创建倒序遍历：

```
public IEnumerable<T> BottomToTop {
        get {
                for (int i = 0; i < count; i++) {
                        yield return items[i];
                }
 }}
```

迭代器中的yield语句

- 使用yield return 产生枚举元素：

```
for (int i = count – 1; i >= 0; --i) {
        yield return items[i];
}
```

- 使用yield break 中断迭代：

```
for (int i = count – 1; i >= 0; --i) {
        yield return items[i];
        if (items[i]>10)
                yield break;
}
```

Agenda

- 使用匿名方法
- 匿名方法机制
- 使用迭代器
- 迭代器机制
- 讲座总结
- Q&A

迭代器机制

- C# 2.0中的迭代器同样是通过编译器的一层额外处理，来简化创建可用于foreach的枚举集合的工作。

- 通过ILDasm.exe反汇编工具，我们可以获得对迭代器的深入理解：
  - 迭代器中的GetEnumerator()方法
  - 迭代器中的嵌套类
  - 迭代器中的yield语句

代码演示

使用ILDasm剖析迭代器机制

Agenda

- 使用匿名方法
- 匿名方法机制
- 使用迭代器
- 迭代器机制
- 讲座总结
- Q&A

讲座总结

- 匿名方法允许我们以一种“内联”的方式将代码直接与委托实例相关联，从而使得委托实例化的工作更加直观和方便。迭代器允许我们更加方便地编写应用于foreach语句的枚举集合。

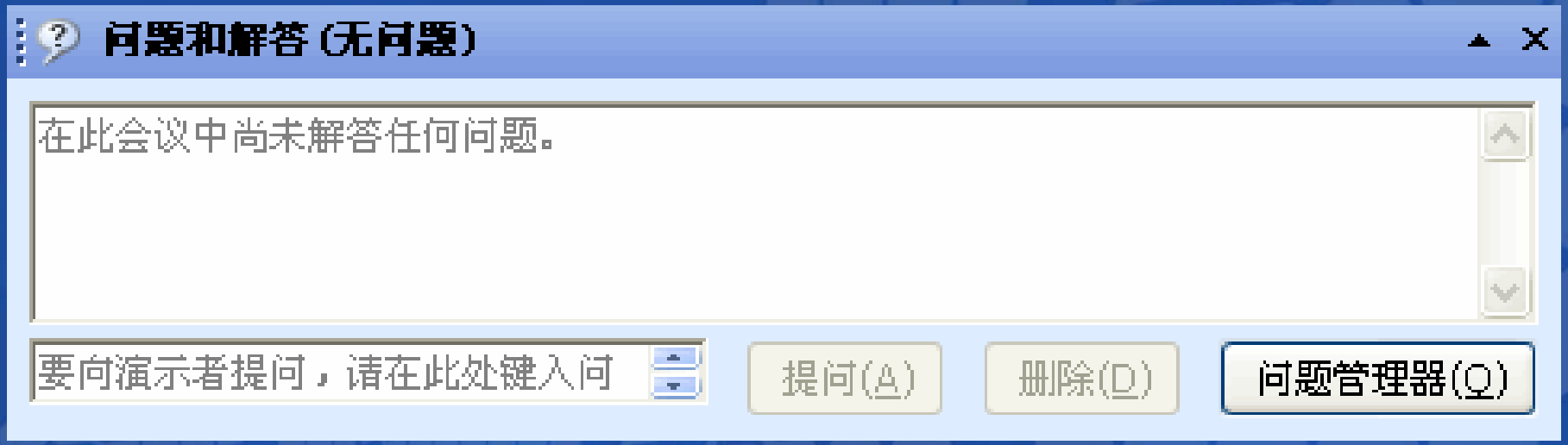
- 对于类似于C#这样的高级语言，掌握好它一个很重要的地方就是掌握编译器在背后为我们做了哪些工作。 C# 2.0中的匿名方法和迭代器都是通过在编译层面进行一些额外的处理，进而来简化程序员的工作。

Agenda

- 使用匿名方法
- 匿名方法机制
- 使用迭代器
- 迭代器机制
- 讲座总结
- Q&A

Q&A

如需提出问题，请单击“提问”按钮并在随后显示的浮动面板中输入问题内容。一旦完成问题输入后，请单击“提问”按钮。




您的潜力，我们的动力
Microsoft
微软（中国）有限公司
Microsoft
msdn
MSDN Webcasts