Lambda表达式基础

北京理工大学计算机学院
金旭亮

委托的典型用法：TypicalDelegateExample示例

1 定义一个委托：

```
public delegate int AddDelegate(int i, int j);
```

2 定义一个MyClass类，放置一个满足AddDelegate委托要求的方法

```
class MyClass
{
    public int Add(int i , int j)
    {
        return i + j;
    }
}
```

3 定义一个委托变量

```
AddDelegate del = null;
```

4 实例化一个MyClass对象，并将其Add方法引用传给委托变量del

```
MyClass obj = new MyClass();
del = obj.Add ;
```

5 通过委托变量调用MyClass对象的Add()方法

```
int result = del(100, 200);
Console.WriteLine(result);//300
```

这太麻烦了！简化一下？

为了简化委托编程，C#首先引入了一种语法特性——**匿名方法**，之后，再自然地演化为了**Lambda表达式**。

匿名方法的示例（ AnonymousMethodToLambda ）

定义委托

```
public delegate int AddDelegate(int i,int j );
```

使用匿名方法定义委托变量

```
AddDelegate del = delegate(int i, int j)
            {
                return i + j;
            };
```

匿名方法“没有名字”，可以直接赋值给委托类型的变量。

通过委托变量“间接”地调用方法

```
Console.WriteLine(del(100,200)); //300
```

一个接收委托类型变量作为参数的方法：

```
public static void invokeDelegate(AddDelegate del ,int i,int j)
{
        Console.WriteLine( del(i, j));
}
```

匿名方法作为方法实参：

```
invokeDelegate( delegate(int i, int j)
                {
                    return i + j;
                }, 100, 200);
```

匿名方法其实是**将方法定义与委托变量赋值两个步骤合而为一**，从而省去了专门定义符合委托要求的方法这个步骤。

更进一步，将匿名方法转换为Lambda表达式.....

```
invokeDelegate((i, j) => i + j , 100, 200);
```

```
(i, j) => i + j
```

```
delegate(int i, int j)
{
    return i + j;
}
```

Lambda表达式在功能上等价于匿名方法，可以看成是匿名方法的进一步简化

Lambda表达式与泛型委托

```
//定义泛型委托
public delegate T MyGenericDelegate<T>(T obj);
```

```
//使用匿名方法给泛型委托变量赋值
MyGenericDelegate<int> del = delegate(int vlaue)
            {
                return vlaue * 2;
            };
```

泛型委托的用法与普通委托没什么两样，只需要在定义时指定泛型参数即可。

```
//调用泛型委托变量引用的匿名方法
Console.WriteLine(del(100));  //200
```

同样可以使用匿名方法或Lambda表达式给泛型委托变量赋值

```
 //使用Lambda表达式给泛型委托变量赋值
MyGenericDelegate<int> del2 = (value) => value * 2;
```

```
//调用泛型委托变量引用的Lambda表达式
Console.WriteLine(del2(100)); //200
```

示例： GenericDelegateAndLambda

.NET基类库中的预定义委托

为了方便软件工程师，.NET基类库中针对在实际开发中最常用的情形提供了几个预定义好的委托，这些委托可以直接使用，无需从头定义自己的委托类型……

`Func<T,…>`	有返回值的方法	`Action<T,…>`	无返回值的方法
`Comparison<T>`	完成对象比较	`Predicate<T>`	完成对象过滤

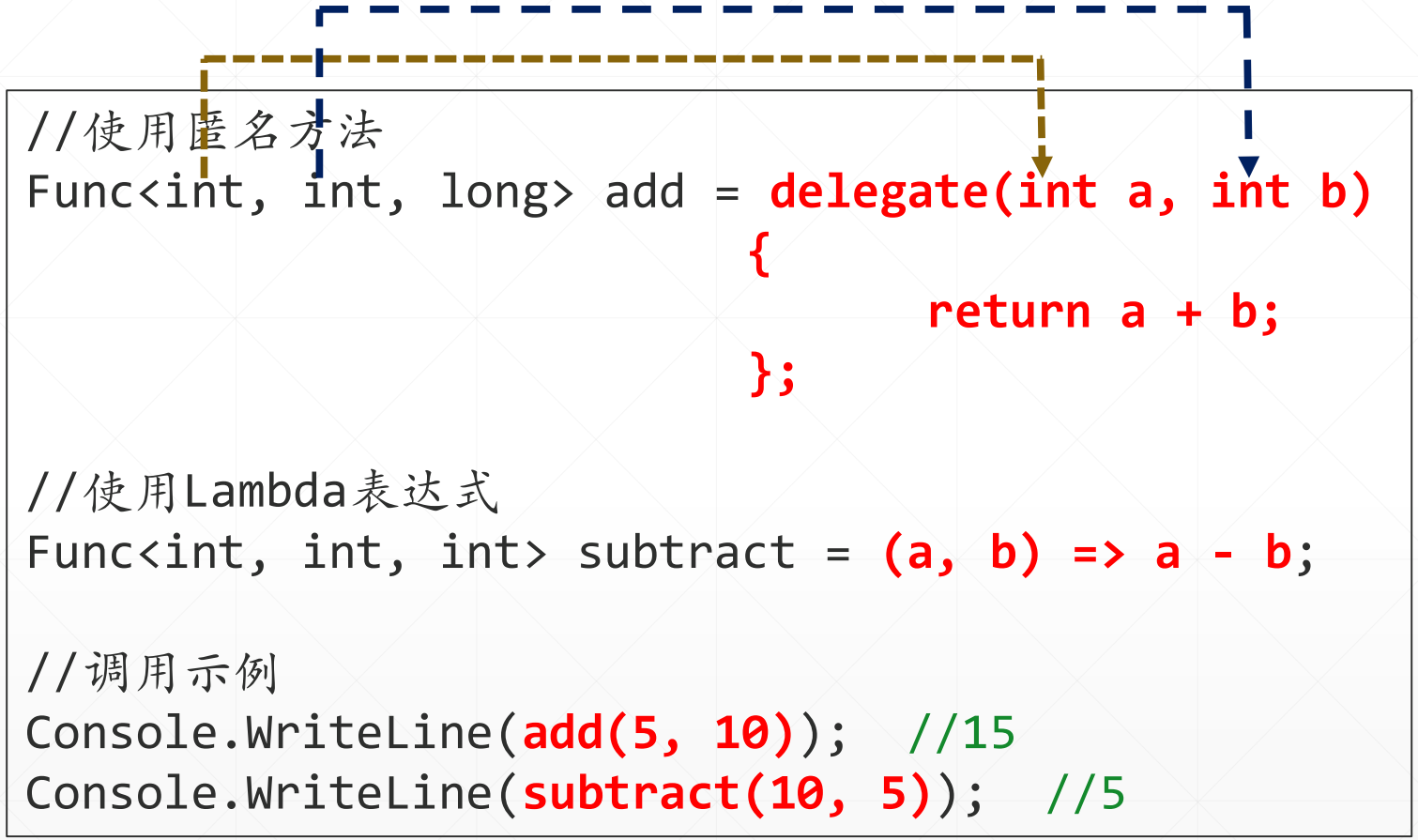
Func<>委托

```
1. public delegate TResult Func< TResult>();
2. public delegate TResult Func<T, TResult>(T arg)
3. public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2)
4. public delegate TResult Func<T1, T2, T3, TResult>(
                                T1 arg1, T2 arg2, T3 arg3)
5. public delegate TResult Func<T1, T2, T3, T4, TResult>(
                                T1 arg1, T2 arg2, T3 arg3, T4 arg4)
```

所有这些看上去复杂的声明，其实是很简单的：

最后一个参数是委托所接收方法的返回值，前面的参数（如果有的话）就是委托所接收方法的形参。

Func<>应用实例



```
//使用匿名方法
Func<int, int, long> add = delegate(int a, int b)
                           {
                                   return a + b;
                           };

//使用Lambda表达式
Func<int, int, int> subtract = (a, b) => a - b;

//调用示例
Console.WriteLine(add(5, 10));  //15
Console.WriteLine(subtract(10, 5));  //5
```

使用.NET基类库（Base Class Library）中的预定义委托，可以省去反复定义委托类型的麻烦。

示例： GenericDelegateInBaseClassLibrary

Func<>泛型委托所能接收的方法都有返回值，那么，有没有接收返回void的方法的系统预定义委托呢?

Action<>委托

```
1. public delegate void Action();
2. public delegate void Action<T>(T obj);
3. public delegate void Action<T1, T2>(T1 arg1,T2 arg2)
4. public delegate void Action<T1, T2, T3>(
                          T1 arg1,T2 arg2,T3 arg3)
5. public delegate void Action<T1, T2, T3, T4>(
                          T1 arg1,T2 arg2,T3 arg3,T4 arg4)
```

上述定义的委托，接收拥有一个到四个参数，返回值类型为**void**的方法。

Action<>实例

```
//使用Lambda表达式给Action委托变量赋值
Action<string> show = (info) =>
                      {
                          Console.WriteLine(info);
                      };

//调用委托变量
show("Hello");
```

示例：GenericDelegateInBaseClassLibrary

用于对象比较的泛型委托

Comparison<T>委托引用的方法可用于比较两个对象的大小。

```
public delegate int Comparison<T>(T x,T y)
```

```
class MyClass
{
   public int Value;
   public string Information;
}
```

示例：UseComparisonDelegate

```
Comparison<MyClass> cmp = (obj1,  obj2)=>
            {
                if (obj1.Value > obj2.Value)
                    return 1;
                else
                    if (obj1.Value < obj2.Value)
                        return -1;
                    else
                        return 0;
             };
```

```
List<MyClass> lst = ……;
lst.Sort(cmp);
```

用于查找对象的委托

```
public delegate bool Predicate<T>(T obj);
```

Predicate<T>委托接收一个方法引用，在方法内部书写代码表明此对象应满足的条件，满足返回true。

```
//定义对象查询条件
Predicate<MyClass> pred = (elem)=>
{
    if (elem.Value > 50)
        return true;
    else
        return false;
};
```

```
List<MyClass> lst = ……;
//在集合中查询对象
MyClass foundElement= lst.Find(pred);

if (foundElement != null)
    //找到了符合条件的对象……
else
   //未找到符合条件的对象……
```

示例： UsePredicateDelegate

把握Lambda表达式的基本语法

编写Lambda表达式的基本格式

如果方法代码中仅包容一条return 语句……

（参数列表）=>表达式

如果方法代码中仅包容多条语句……

（参数列表）=> {语句1;语句2;……}

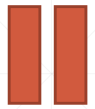
“=>” 念做“goes to”。

示例： LambdaSyntax

学会编写Lambda表达式-1

只有一个输入参数时，括号是可选的：

```
Func<int, bool> del = x => x > 0;
```



```
Func<int, bool> del = (x) => { return x > 0; };
```

两个或更多输入参数由括在括号中的逗号分隔，**参数名字无关紧要**。

```
Func<int, int, bool> del = (x, y) => x == y;
```

Lambda表达式中的return关键字可省。

学会编写Lambda表达式-2

对于Lambda表达式的参数，在声明时可以不指定数据类型，而由编译器来动态地进行类型推断。

有时，出于代码易读性的考虑，也可以显式指定类型：

```
Func<int,string,bool> del = (int x, string s) => s.Length > x;
```

如果没有参数，则使用空括号

```
Action del = () => { Console.WriteLine("No arguments"); };
```