

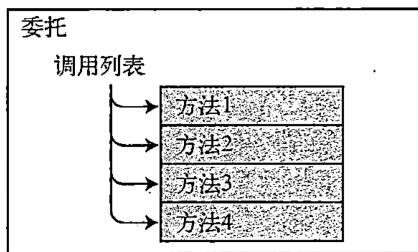
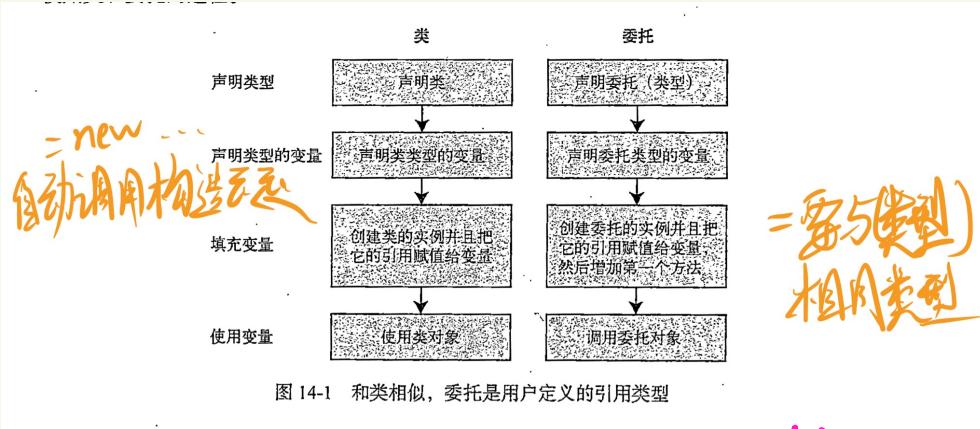
07 05 23

北島

C# 図解 第十四章

李 托

流程与 diff 类



调用方法包含在 delegate ?
方法可 static 也可非 static (即静态方法)
只要委托返回类型相同?
基本匹配元 (除了多态类型, 例如)

委托类型声明在两个方面与方法声明不同。委托类型声明：

- 以 delegate 关键字开头；
- 没有方法主体。

声明变量之后加?

说明 虽然委托类型声明看上去和方法的声明一样，但它不需要在类内部声明，因为它是类型声明。

14. b/T1 组合与添加 (del(创建后不可改变)与移除
del C = delA + delB 组合

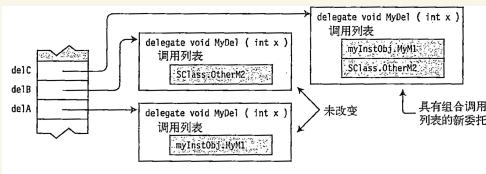


图 14-6 组合委托

添加：“+ =”

```
MyDel delVar = inst.MyM1; // 创建并初始化
delVar += SC1.m3; // 增加方法
delVar += X.Act; // 增加方法
```

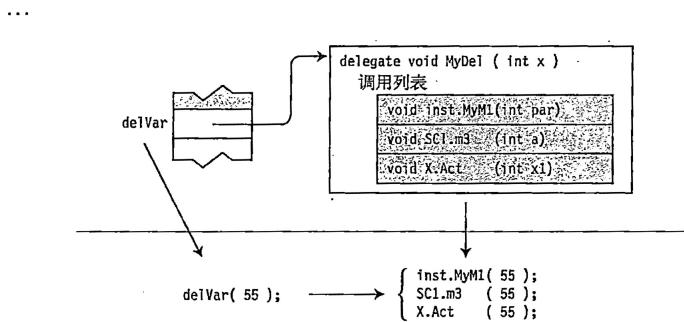
且从上往下增加调用列表

实际上每次“+ =”之后都是引用指向新的del, 原del free!

移除 “-=” like “+=” 创建新副本
当方法有多个实例，则会从列表最末开始
每用坏掉在的先被调用空委托抛异常

14.9 调用:

```
if (delVar != null)  
{ delVar(55); } //调用委托  
delVar?.Invoke(65); //使用 Invoke 和空条件运算符
```



感觉?.invoke
但else怎么写?
better



14.11 带返回值

如果委托有返回值并且在调用列表中有一个以上的方法，会发生下面的情况。

- 调用列表中最后一个方法返回的值就是委托调用返回的值。
- 调用列表中所有其他方法的返回值都会被忽略。

14.12 带引用参数：

如果委托有引用参数，参数值会根据调用列表中的一个或多个方法的返回值而改变。

在调用委托列表中的下一个方法时，参数的新值（不是初始值）会传给下一个方法。例如，如下代码调用了具有引用参数的委托。图 14-11 演示了这段代码。

如果引值类型也许不变？

14.13 匿名方法

针对该语法只用来实现委托
匿名方法返回类型自动适配委
托需要的返回类型

class Program

```
{  
    public static int Add20(int x)  
    {  
        return x + 20;  
    }
```

```
delegate int OtherDel(int InParam);  
static void Main()  
{  
    OtherDel del = delegate(int x)  
    {  
        return x + 20;  
    };  
    Console.WriteLine("{0}", del(5));  
    Console.WriteLine("{0}", del(6));  
}
```

等效

② 匿名方法参数需与委托匹配

矛盾，但个人认为麻烦

2. 参数

除了数组参数，匿名方法的参数列表必须在如下 3 方面与委托匹配：

- 参数数量；
- 参数类型及位置；
- 修饰符。

可以通过使圆括号为空或省略圆括号来简化匿名方法的参数列表，但必须满足以下两个条件：

- 委托的参数列表不包含任何 out 参数；
- 匿名方法不使用任何参数。

例如 加下代码声明了一个没有 out 参数的委托 和一个没有使用任何参数的匿名方法

我决定不修改

(3) params 参数的 params 关键字
params int[] Y \Rightarrow int[] Y

14. 13. 3 作用域

参数以及声明在匿名方法内部的局部变量的作用域限制在实现代码的主体之内，

匿名方法不可访问其外层作用域的变量 (外部变量)

用圆圈画、实现外部变量符被方法捕获

生命周期扩展：

栈与局部
代码块

The diagram shows a code snippet with annotations:

```
delegate void MyDel();
static void Main()
{
    MyDel mDel;
    int x = 5;
    mDel = delegate
    {
        Console.WriteLine("Value of x: {0}", x);
    };
    // Console.WriteLine("Value of x: {0}", x);
    if (null != mDel)
        mDel();
}
```

- 变量x定义在外部块中，在匿名方法之外。
- x的作用域：一个大括号包围了匿名方法和if语句块。
- 变量x被匿名方法捕获：指向匿名方法体内的x声明。
- 变量x离开了作用域并会导致编译错误：指向if语句块外的x引用。
- 而这里使用了x，
在匿名方法内部：指向匿名方法体内的x声明。

图 14-15 在匿名方法中捕获的变量

局部变量 x 的流域性

14.14 Lambda 表达式（3.0 之后代替了 2.0 的匿名方法）
So, 上面 13 处拿掉为了解释 Lambda 吗？

托。我们可以很容易地通过如下步骤把匿名方法转换为 Lambda 表达式：

- 删除 delegate 关键字；
- 在参数列表和匿名方法主体之间放置 Lambda 运算符=>。Lambda 运算符读作“goes to”。

```
MyDel del = delegate(int x) { return x + 1; };      //匿名方法
MyDel le1 =      (int x) => { return x + 1; };      //Lambda 表达式
```

编译器还可以从委托的声明中知道委托参数的类型，因此 Lambda 表达式允许省略类型参数，如 le2 的赋值代码所示。

带有类型的参数列表称为显式类型。

省略类型的参数列表称为隐式类型。

如果只有一个隐式类型参数，我们可以省略两端的圆括号，如 le3 的赋值代码所示。

最后，Lambda 表达式允许表达式的主体是语句块或表达式。如果语句块包含了一个返回语句，我们可以将语句块替换为 return 关键字后的表达式，如 le4 的赋值代码所示。

```
el del = delegate(int x) { return x + 1; } ; //匿名方法  
el le1 = (int x) => { return x + 1; } ; //Lambda 表达式  
el le2 = (x) => { return x + 1; } ; //Lambda 表达式  
el le3 = x => { return x + 1; } ; //Lambda 表达式 2 due to just one arg! /省去()  
el le4 = x => x + 1 ; //Lambda 表达式 d lu hui?
```

省去参数类型 due to 委托声明中写了
2 due to just one arg! /省去()
d lu hui?

替代匿名，若具备则无法替代

方法为一个代码跳转执行

委托是一个类型的

choose whi

But I still can't understand